Fee-Based Spam Prevention For Lightning

John Law

May 14, 2025

Version 1.2

Abstract

This paper presents protocols for assigning and collecting fees for Lightning services, thus reducing the potential for spam on the Lightning Network. The protocols determine and collect fees for all significant costs that are incurred. While previous proposals have attempted to reduce spam by charging fees for delayed and/or failed payments, those proposals have failed to reimburse certain costs or have relied on trust to ensure that fees are not stolen.

The collection of fees presented here relies on *griefer-penalization*. That is, a party can be made to lose certain funds by a channel partner who griefs them, but in such a case the griefer also has to lose a comparable amount of funds. Therefore, a party that only selects self-interested partners will never be griefed.

1 Overview

In order to make a payment on the Lightning Network (LN), the sender attempts to route the payment to its destination via a series of two-party Lightning channels where the downstream party in one channel is the upstream party in the next channel. At each channel, the two parties that own the channel attempt to create a Hash Time-Locked Contract (HTLC) in which the upstream party agrees to pay the downstream party a fixed amount if the downstream party provides a specified secret by the HTLC's expiry. The expiries of the HTLCs are staggered so that a party that receives a secret and pays an HTLC in one channel has the time to provide that secret and obtain payment in the preceding channel. In addition, the values of the HTLCs are structured so that they transfer the payment's funds plus fees to the parties that successfully routed the payment. As a result, routing nodes are rewarded with fees whenever they participate in a successful Lightning payment.

However, routing nodes do not receive any fees for a payment that fails. This is problematic, as the routing nodes have to perform calculations, exchange messages, and devote scarce resources (including

channel funds) to the payment, even if it fails. As a result, routing nodes can be subject to spam attacks in which they devote their resources to payments that are intentionally designed to fail.

The LN's vulnerability to spam attacks has been acknowledged for many years and researchers have proposed several protocols for reducing spam on Lightning **[Jager20][RN22][ST22]**. In addition, even if no one intentionally causes payments to fail, the disconnect between the consequences of one's actions and the rewards or penalties for those actions makes it likely that the LN's resources will be poorly utilized. For example, a routing node that delays a successful payment pays no price as long as the delaying node's resources are not required for any other payments, even if the delay limits other nodes' ability to route other payments and inconveniences the payment's sender.

In summary, the economic incentives in the current LN make the network susceptible to spam attacks and fail to reward the most efficient use of resources.

This paper presents protocols that build on ideas created by Jager **[Jager20][Jager21]**, Teinturier **[Teint20a][Teint20b]**, Towns **[Towns23]**, and Riard and Naumenko **[Riard22][RN22]**. These protocols assign and collect fees such that any party who imposes a significant cost on another is forced to fully reimburse the affected party. As a result, these protocols should improve the efficiency of the LN and reduce spam on the network.

The collection of fees presented here relies on *griefer-penalization*. That is, a party can be made to lose certain funds by a channel partner who griefs them, but in such a case the griefer also has to lose a comparable amount of funds. Therefore, a party that only selects self-interested partners will never be griefed.

The rest of this paper is organized as follows. The types of fees are presented in Section 2 and the collection of fees is covered in Section 3. Sections 4 and 5 present protocols for calculating fees. Sections 6 analyzes the risk of fee loss and Section 7 discusses related work. Conclusions are given in Section 8. The appendix contains detailed descriptions of the protocols.

2 Fee Types

The protocols presented here collect 3 types of fees:

- an *Upfront Fee* paid by the sender to the downstream node in each channel which updates the channel state to include an HTLC for the sender's Lightning payment,
- a *Hold Fee* paid by any node that delays the payment to each upstream node that they delay, and
- a *Success Fee* paid by the sender to each node that helps deliver a successful Lightning payment.

The Upfront Fee pays for:

- the computation and communication costs of routing the payment,

- the cost of allocating a Commitment transaction output ("slot") to the payment[1],
- the cost of allocating channel funds to the payment for the seconds that can be required for a non-delayed payment,
- the risk of not fulfilling the HTLC with one's upstream partner despite one's downstream partner having fulfilled their HTLC[2],
- the risk of having to pay a Hold Fee (as described below), and
- the risk of burning funds (as described below).

Before routing or receiving a Lightning payment, each node publishes its *hold_grace_period_delta_msec* parameter giving the minimum time they can take without having to pay for delaying a Lightning payment[3]. If a router or the destination delays a Lightning payment beyond its grace period, it pays a Hold Fee to:

- each of the upstream nodes for the cost of their capital held for the length of the delay, and
- the sender for the inconvenience caused by the delay.

The Success Fee pays for the service of successfully completing a payment. As a result, it motivates the routers and the destination to make the payment attempt succeed.

3 Fee Collection

3.1 Success Fees

As in the current Lightning protocol, the Success Fee is paid by increasing the size of each HTLC in order to include fees for the downstream node in the current channel and in all later channels. As a result, any node that participates in a successful payment will be rewarded with a Success Fee.

3.2 Upfront And Hold Fees

Burn Output

The collection of Upfront and Hold Fees relies on the addition of a *burn output* to the Lightning channel's transactions. The burn output can be spent by anyone after a 20-year delay (so the burn output will be claimed by a miner 20 years in the future). Both parties in a Lightning channel put some of their funds in the burn output, and when the payment's HTLC is resolved the protocol determines how the

1 If the channel protocol requires the allocation of an output to each payment. The current Lightning protocol requires such an allocation, but the Off-chain Payment Resolution (OPR) protocol **[Law24]** does not.

2 And thus having to pay an HTLC while not receiving payment for the corresponding HTLC.

3 Unless the payment uses the Off-chain Payment Resolution (OPR) protocol **[Law24]**, in which case the *hold_grace_period_delta_msec* parameter is replaced by the *htlc_expiry_delta_msec* parameter and no Hold Fees are paid (as payments cannot be delayed when using the OPR protocol).

burn output's funds are divided between the channel partners. The protocol allows both parties to calculate the same division of the burn output's funds, and the fact that those funds will be burned unless the parties agree on their division incentivizes cooperation.

Terminology

The following terminology will be used for the collection of the Upfront and Hold Fees:

- a routing node is ***paid*** a fee for providing routing services,
- a routing node ***stakes*** funds to pay for other nodes' fees by placing the staked funds in a burn output, and
- staked funds are divided between the parties in a channel by ***refunding*** some or all of them to the party that staked them and ***transferring*** the remainder to their channel partner.

Whenever an HTLC for a Lightning payment is created for a channel:

- the upstream channel partner moves *upfront_stake_msat* msats from their output to the burn output to cover the maximum Upfront Fees that could be paid to all downstream nodes,
- the downstream channel partner moves *hold_stake_msat* msats from their output to the burn output to cover the maximum Hold Fees that could be paid to all upstream nodes, and
- each node moves funds (called *matching funds*) from their output to the burn output that are a fixed fraction of both parties' burn contributions for Upfront and Hold Fees.

The matching funds ensure that both parties have an interest in coming to agreement on the correct division of the burn output's funds **[Law24]**.

Notation Used In Figures

Throughout this paper, protocols will be illustrated with figures where:

- **A** denotes Alice's signature,
- **B** denotes Bob's signature,
- **AB** denotes both Alice's and Bob's signature,
- **revkey{A|B}i** denotes {Alice's|Bob's} revocation key signature that revokes the other party's state *i* transaction,
- **pckey{A|B}i** denotes a signature using a per-commitment key for revoking {Alice's|Bob's} state *i* transaction,
- ***tsd*{A|B}** denotes the *to_self_delay* channel parameters set by {Alice|Bob}, and

- **Preimage(X)** denotes the payment secret (and payment receipt) which is the preimage of X.

Shaded boxes represent transactions that are on-chain and unshaded boxes represent off-chain transactions. Each box includes a label showing the transaction type, namely:

- **F** for the Funding transaction,
- **Com** for a Commitment transaction,
- **H-t** for an HTLC-timeout transaction,
- **H-s** for an HTLC-success transaction,
- **Pay** for a Payment transaction,
- **In** for an Individual transaction,
- **St** for a State transaction,
- **H-k** for an HTLC-kickoff transaction, and
- **H-p** for an HTLC-payoff transaction.

Subscripts denote which party can put the transaction on-chain (if only one party can do so) and which channel state the transaction is associated with (namely state *i* in the figures). Transactions that can only be put on-chain by Alice are green, those that can only be put on-chain by Bob are red, and those that can be put on-chain by either party are black.

Bold lines carry channel funds, thin solid lines have value equal to (or slightly larger than[4]) the tunable penalty amount, and dashed lines have the minimal allowed value (as they are used for control). When a single output can be spent by multiple off-chain transactions, those transactions are said to *conflict*, and only one of them can be put on-chain.

Each output is labeled with the requirements that must be met to spend the output, with multiple cases being shown as outputs that branch. Multiple requirements are indicated with "&" between them and are listed with knowledge of a secret first, relative delays next, then absolute timelocks (in parentheses) and signature requirements last.

Location Of The Burn Output

The location of the burn output depends on the protocol that is being used to maintain the channel state and to resolve Lightning payments.

4 The value of the first output of each State transaction equals the tunable penalty amount, while the output of the Individual transaction has a slightly larger value in order to provide funds for the State transaction's HTLC control outputs.

If the current Lightning protocol is being used, the burn output can be added to each party's Commitment transaction as shown in Figure 1.

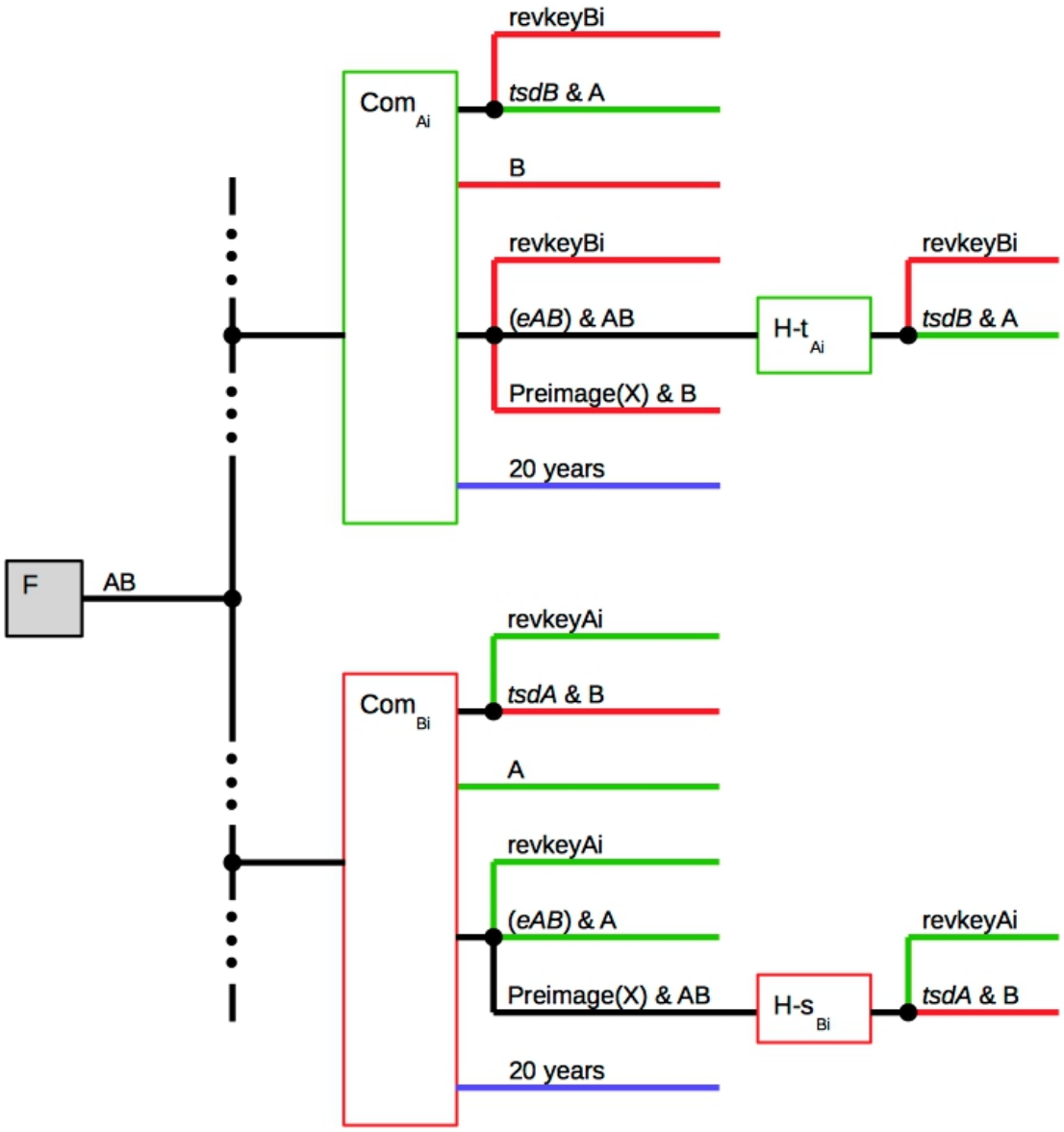

Figure 1. Current Lightning protocol showing an HTLC offered by Alice to Bob. A burn output for fees (payable to anyone in 20 years) has been added to each party's Commitment transaction.

This placement is reasonable as long as the size of the burn output remains small relative to each of the party's outputs[5]. On the other hand, if the burn output is large relative to Alice's output, Alice can grief

5 Upfront and Hold Fees should be small relative to HTLC payment amounts, so requiring each party to maintain even a small channel balance should be sufficient to assure that the burn output is small relative to each party's output.

Bob by putting an old channel state with a large burn output on-chain. Even though Bob can use the old state's revocation key to obtain Alice's funds from that state, Bob cannot obtain the funds in the old state's burn output. If this is a potential problem, each party can be required to use a separate Payment transaction in order to spend the *to_self* output of their Commitment transaction as shown in Figure 2[6].

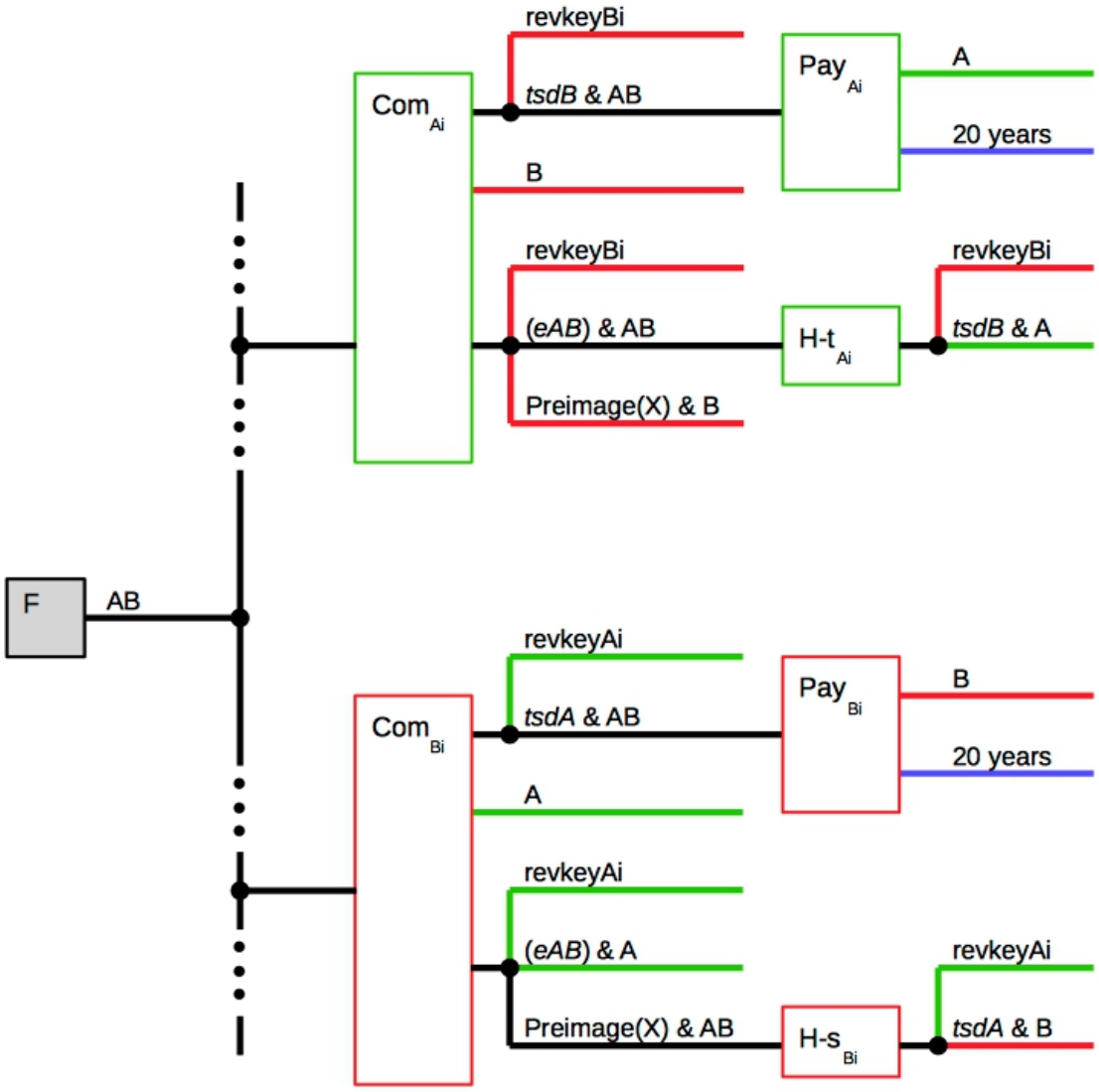

Figure 2. Current Lightning protocol showing an HTLC offered by Alice to Bob. A burn output has been added to a Payment transaction that is used to spend the Commitment transaction's *to_self* output.

If some form of Tunable-Penalty protocol such as the Fully-Factory-Optimized (FFO) **[Law22]** protocol is used, the burn output is added to the current Commitment transactions as shown in Figure 3.

6 A similar technique was presented in Section 9.1 of **[Law24]**.

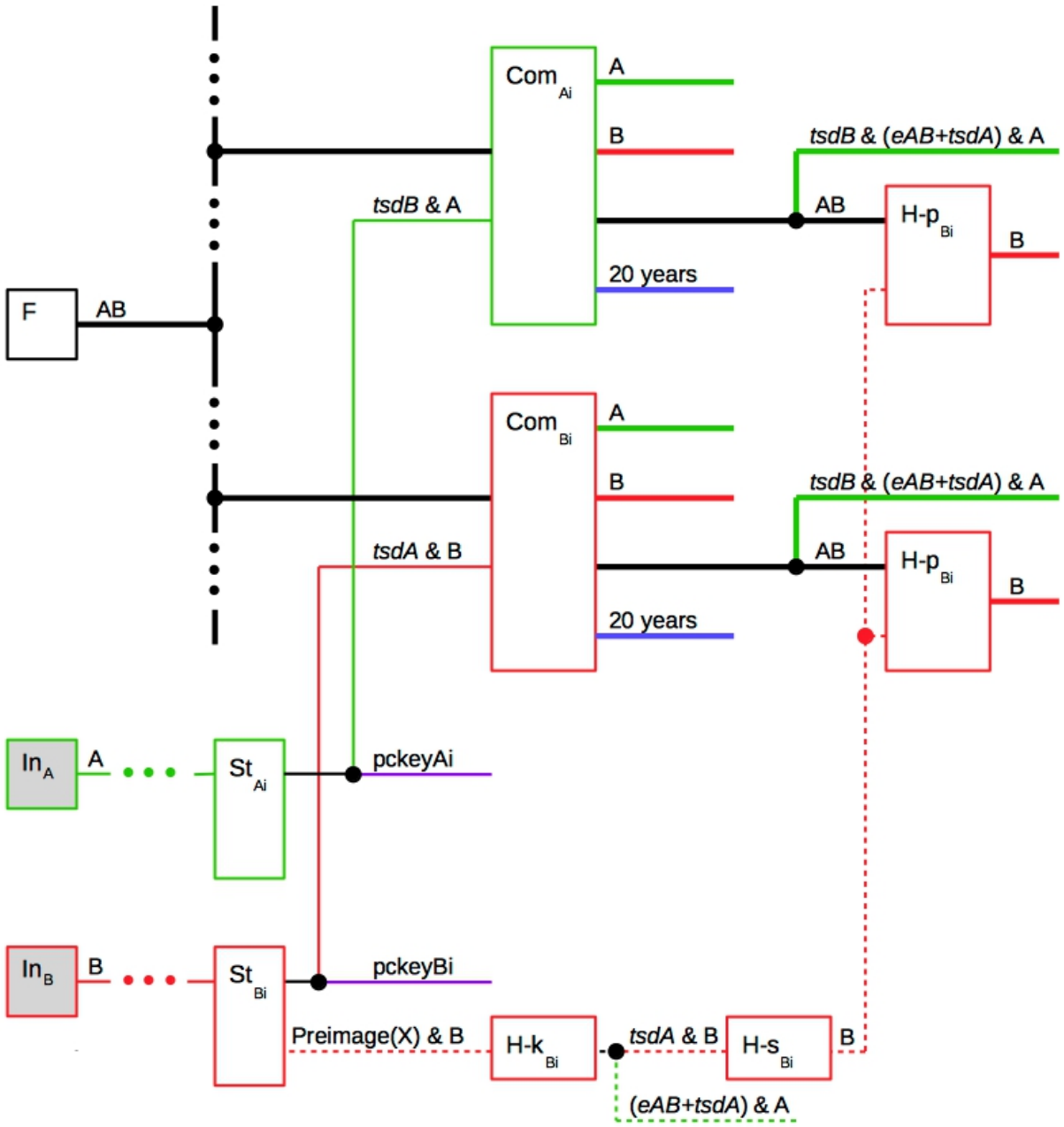

Figure 3. Fully-Factory-Optimized (FFO) protocol showing an HTLC offered by Alice to Bob. A burn output has been added to each party's Commitment transaction.

Finally, Figures 4 and 5 show that a single burn output holds funds for both the HTLC and the Upfront and Hold Fees when using the Off-chain Payment Resolution (OPR) **[Law24]** protocol.

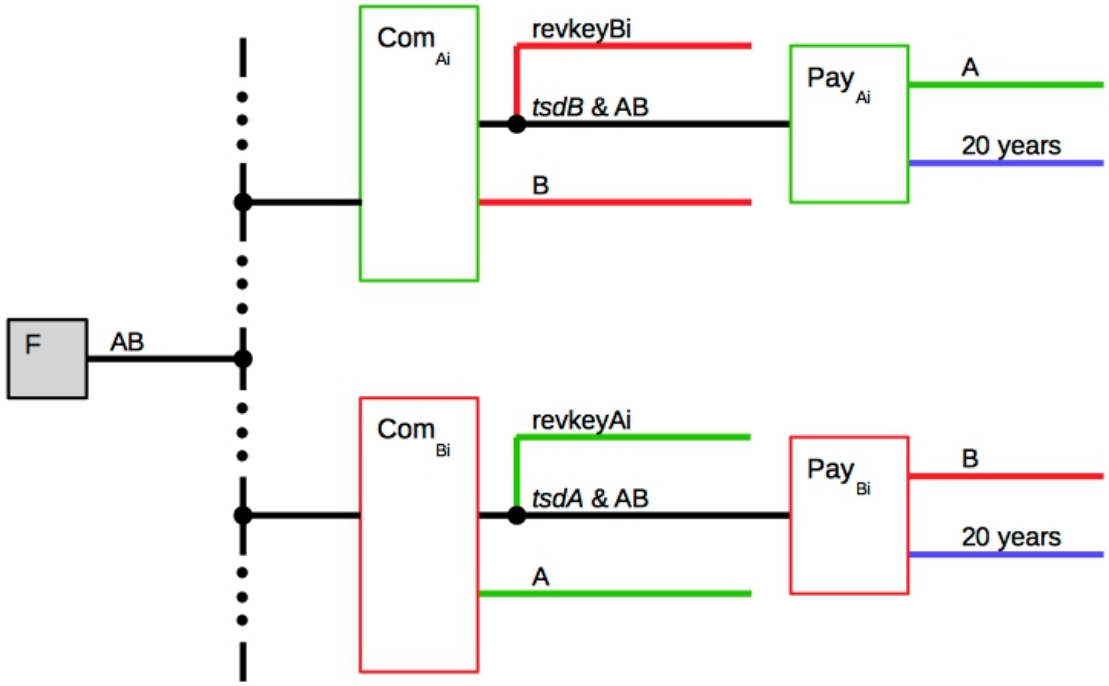

Figure 4. An HTLC offered by Alice to Bob using the current Lightning channel protocol and OPR payment resolution protocol. The same burn output is used for both the HTLC and the Upfront and Hold Fees.

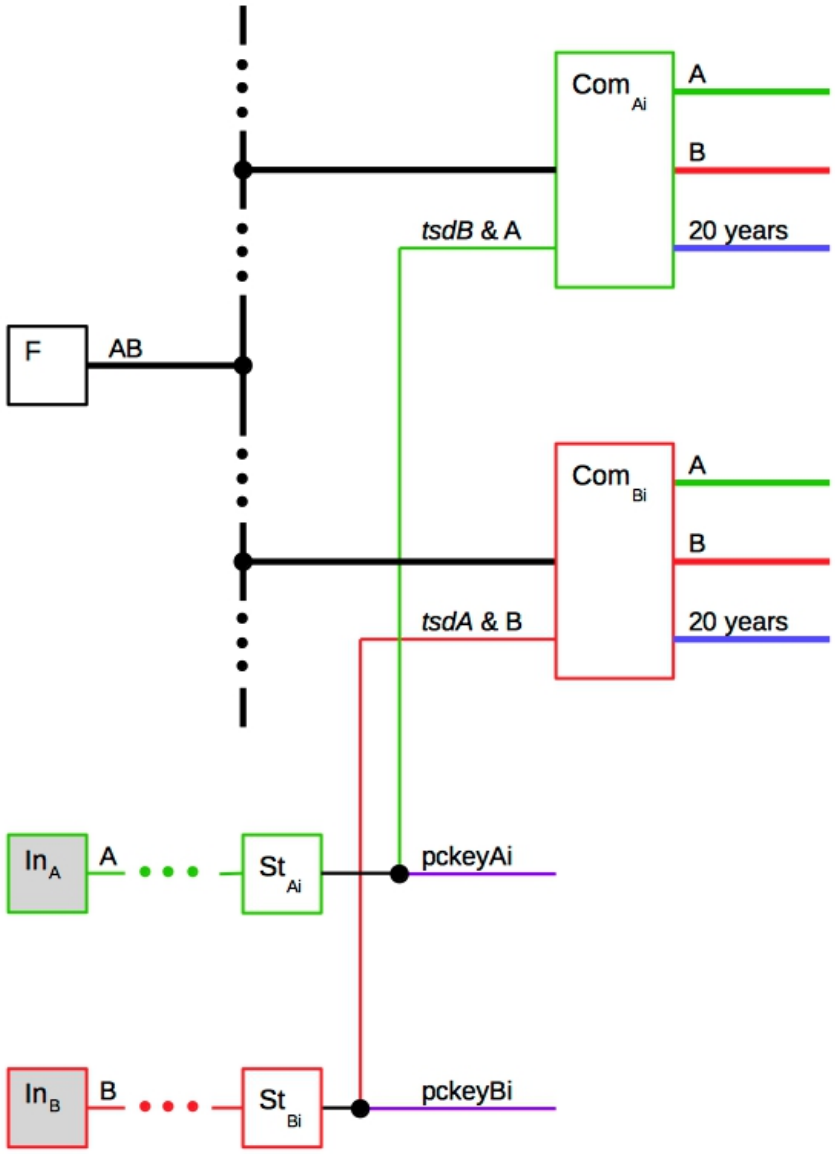

Figure 5. An HTLC offered by Alice to Bob using the FFO channel protocol and OPR payment resolution protocol. The same burn output is used for both the HTLC and the Upfront and Hold Fees.

Ordering Of Updates To The Channel State

The order in which updates to the channel state are made depends on the protocol that is being used to maintain the channel state and to resolve Lightning payments.

If the OPR protocol is being used to resolve the payment (as in Figures 4 and 5 above) funds for the HTLC and the Upfront Fee are placed in a single burn output (and there is no Hold Fee). As a result, the movement of funds to the burn output can be implemented with the same sequence of message

transfers as in the current Lightning channel protocol. Specifically, the movement of funds to the burn output uses the flow from the BOLT 02 spec **[BOLT2]**:

1. pending on the downstream node
2. ... in the downstream node's latest commitment transaction
3. ... and the downstream node's previous commitment transaction has been revoked, and the update is pending on the upstream node
4. ... and in the upstream node's latest commitment transaction
5. ... and the upstream node's previous commitment transaction has been revoked

On the other hand, if on-chain payment resolution is used (as in Figures 1, 2 and 3 above) creating a new HTLC requires both increasing the burn amount and adding an HTLC output. If these two changes were made together using the flow from the BOLT 02 spec shown above, the downstream node would be able to force the upstream node to pay an unreimbursed Hold Fee **[Hard25]**.

To see this, consider the case where Alice offers a new HTLC to Bob and only steps 1 and 2 of the flow above are performed. In this case, Bob has two sets of transactions, one of which does not include updates for the new HTLC and the other of which includes both the increased burn amount and the HTLC output for the new HTLC. If Bob does not revoke his old state before the grace period for the new HTLC expires, Alice will be responsible for paying a Hold Fee in the payment's previous channel for any further delay. However, Alice cannot fail the HTLC in the previous channel, as Bob has the ability to claim payment for the HTLC on-chain. Furthermore, Alice cannot be sure of being reimbursed by Bob for any Hold Fee, as Bob has the ability to put his old state (without the increased burn amount) on-chain. Therefore, Alice can be forced to pay an unreimbursed Hold Fee in the previous channel until either Bob responds off-chain or Alice resolves her HTLC with Bob on-chain. Bob's ability to force Alice to pay an unreimbursed Hold Fee even though Alice did not delay the payment violates the goal of having each party pay for the costs that they impose on others.

The solution to this problem is to separate the update of the channel state for adding a new HTLC into two steps, namely increasing the burn amount first and then adding the HTLC output. The most straightforward way to implement these two steps is to use the following flow for adding an HTLC:

1. increased burn amount is pending on the downstream node
2. ... and the increased burn amount is in the downstream node's latest commitment transaction
3. ... and the downstream node's previous commitment transaction has been revoked, and the increased burn amount is pending on the upstream node
4. ... and the increased burn amount is in the upstream node's latest commitment transaction
5. ... and the upstream node's previous commitment transaction has been revoked, and the HTLC output is pending on the downstream node

6. ... and the increased burn amount and HTLC output are in the downstream node's latest commitment transaction
7. ... and the downstream node's previous commitment transaction has been revoked, and the HTLC output is pending on the upstream node
8. ... and the increased burn amount and HTLC output are in the upstream node's latest commitment transaction
9. ... and the upstream node's previous commitment transaction has been revoked

With this flow if Alice offers Bob an HTLC and Bob does not commit to being responsible for a Hold Fee (step 3) before the HTLC's grace period has expired, Alice can fail the payment in the payment's previous channel. When Bob's revocation of his previous state is finally received by Alice (step 3), Alice can fail the payment in the channel with Bob by sending Bob an *update_fail_htlc* message which prevents an output from being added for this HTLC. The flow in this case is:

1. increased burn amount is pending on the downstream node
2. ... and the increased burn amount is in the downstream node's latest commitment transaction
3. ... and the downstream node's previous commitment transaction has been revoked, and the increased burn amount is pending on the upstream node
4. ... and the increased burn amount is in the upstream node's latest commitment transaction
5. ... and the upstream node's previous commitment transaction has been revoked, and the failed HTLC (without increased burn amount or HTLC output) is pending on the downstream node
6. ... and the failed HTLC (without increased burn amount or HTLC output) is in the downstream node's latest commitment transaction
7. ... and the downstream node's previous commitment transaction has been revoked, and the failed HTLC (without increased burn amount or HTLC output) is pending on the upstream node
8. ... and the failed HTLC (without increased burn amount or HTLC output) is in the upstream node's latest commitment transaction
9. ... and the upstream node's previous commitment transaction has been revoked

While these flows implement the safe collection of Hold Fees, they increase the payment latency from 1.5 round-trip times (RTTs) to 2.5 RTTs.

Fortunately, it is possible to implement the safe collection of Hold Fees without increasing the payment latency. The idea is to increase burn funds before adding the HTLC output (as shown above) but to allow the downstream node to provide signatures for both of these updates consecutively (without a RTT in between). If the downstream node committed to the increased burn funds (by revoking their previous commitment transaction) **during** the grace period, the upstream node can commit to the

addition of the HTLC output (by revoking all earlier commitment transactions). Then the upstream node gives the downstream node a signature for a commitment transaction with the HTLC output. At this point, the downstream node is guaranteed to be able to get payment for the HTLC if they get the hash preimage in time, so the downstream node can offer the HTLC in the payment's next channel. Note that the overall latency is just 1.5 RTTs.

On the other hand, if the downstream node committed to the increased burn funds **after** the grace period, the upstream node does not have to commit to the addition of the HTLC output. Instead, the upstream node sends an *update_remove_htlc* message, as shown above.

This lower-latency protocol requires the upstream node to have up to 3 current (signed and not revoked) transactions at a time. Therefore, the rules for providing *per_commitment_points* and *per_commitment_secrets* have to be modified. The details are provided in the appendix.

The flow for the successful addition of an HTLC using this lower-latency protocol is as follows:

1. increased burn amount is pending on the downstream node
2. ... and the increased burn amount is in the downstream node's latest commitment transaction
3. ... and all of the downstream node's earlier commitment transactions have been revoked, and the increased burn amount is pending on the upstream node
4. ... and the increased burn amount is in the upstream node's latest commitment transaction, and the HTLC output is pending on the upstream node
5. ... and the increased burn amount and HTLC output are in the upstream node's latest commitment transaction
6. ... and the upstream node's earliest current commitment transaction has been revoked
7. ... and the upstream node's previous commitment transaction has been revoked, and the HTLC output is pending on the downstream node
8. ... and the HTLC output is in the downstream node's latest commitment transaction
9. ... and all of the downstream node's earlier commitment transactions have been revoked

However, if the downstream node commits to being responsible for a Hold Fee (step 3) after the grace period, the flow with the lower-latency protocol becomes:

1. increased burn amount is pending on the downstream node
2. ... and the increased burn amount is in the downstream node's latest commitment transaction
3. ... and all of the downstream node's earlier commitment transactions have been revoked, and the increased burn amount is pending on the upstream node

4. ... and the increased burn amount is in the upstream node's latest commitment transaction, and the HTLC output is pending on the upstream node
5. ... and the increased burn amount and HTLC output are in the upstream node's latest commitment transaction
6. ... and the failed HTLC (without increased burn amount or HTLC output) is pending on the downstream node
7. ... and the upstream node's earliest current commitment transaction has been revoked
8. ... and the failed HTLC (without increased burn amount or HTLC output) is in the downstream node's latest commitment transaction
9. ... and all of the downstream node's earlier commitment transactions have been revoked, and the failed HTLC (without increased burn amount or HTLC output) is pending on the upstream node
10. ... and the failed HTLC (without increased burn amount or HTLC output) is in the upstream node's latest commitment transaction
11. ... and the upstream node's earliest current commitment transaction has been revoked
12. ... and the upstream node's previous commitment transaction has been revoked

A detailed description of this lower-latency protocol is given in the appendix.

4 Determining Upfront Fee Stake And Transfer Amounts

4.1 Overview

In each channel the upstream node stakes Upfront Fee funds that are sufficient to pay Upfront Fees to all of the nodes that are downstream of it along the payment's path. The staked Upfront Fees are then divided according to how far the payment is routed.

We will use the following notation and terminology:

- Let ***n*** denote the number of payment hops, with node 0 being the sender and node *n* being the destination.
- For $0 < k \leq n$, a payment ***stops*** at node *k* if node *k* is a downstream node in the last channel which updated its state to include an HTLC for the given payment[7].

This paper will present two protocols for determining Upfront transfer amounts. In both protocols the channel partners create a contract for transferring Upfront Fee funds to the downstream node if that node provides a required secret before the contract's expiry. As a result, the contract for transferring Upfront funds is similar to an HTLC. A key difference, however, is that the contract for transferring

7 If no channel's state was updated to include an HTLC for the given payment, the payment ***stops*** at node 0.

Upfront funds has multiple cases, each with a different transfer amount based on which secret is provided by the downstream node. These different secrets correspond to different nodes at which the payment could stop.

4.2 Preimage Per Stopping Node

The first protocol uses secrets that are hash preimages, with a separate preimage per stopping node.

Notation

This protocol uses the following notation and terminology:

- For $0 < i \le n$, let $\boldsymbol{u_i}$ denote the Upfront Fee paid to node i if it is due an Upfront Fee payment.
- For $0 < i \le n$ and $0 < k \le n$, let $\boldsymbol{t_{i,k}} = \sum_{j=i}^{k} u_j$ denote the amount of Upfront Fee transferred to node i from their upstream partner for a payment that stops at node k[8].
- For $0 < i \le n$, let $\boldsymbol{f_i} = t_{i,n} = \sum_{j=i}^{n} u_j$ denote the Upfront Fee staked by upstream node i-1 for potential transfer to node i.

Thus, if the payment stops at node k, each node i, $0 < i \le k$, receives a transfer of $t_{i,k}$ Upfront funds staked by their upstream partner, and the remaining $f_i - t_{i,k}$ of those staked funds are refunded to the node who staked them. As a result, each node i, $0 < i \le k$, is paid a net of $t_{i,k} - t_{i+1,k} = u_i$ in Upfront Fees.

Protocol

The hop payload of the onion packet received by node i, $0 < i \le n$, includes:

- a secret X_i.

The *update_add_htlc* message sent to node i, $0 < i \le n$, includes:

- the list of pairs $(Y_i, t_{i,i})$, $(Y_{i+1}, t_{i,i+1})$ ... $(Y_n, t_{i,n})$ where, for each k, $i \le k \le n$, $Y_k = hash(X_k)$.

Nodes i-1 and i, $0 < i \le n$, create the following contract:

- node i-1 stakes $f_i = t_{i,n}$ Upfront funds for potential transfer to node i, and
- if node i includes a secret X_k in the *update_fulfill_htlc* or *update_fail_htlc* message that it provides to node i-1, and if $hash(X_k) = Y_k$ for some k, $i \le k \le n$, then node i-1 will transfer $t_{i,k}$ Upfront funds to node i.

8 For ease of explanation, we will focus on the case where the payment stops at some node k, where all nodes up to and including node k execute the protocol without failure. A full analysis requires considering cases where a node fails after it forwards the payment, and where there are failures by multiple nodes.

When it receives an *update_add_htlc* message, node *i* verifies that $t_{i,i} = u_i$ meets its requirements for an Upfront Fee. Node *i* also calculates the list of pairs to include in the *update_add_htlc* message sent to its downstream partner by using the identity $t_{i+1,m} = t_{i,m} - u_i$ for each *m*, $i < m \leq n$. Finally, node *i*-1 stakes $f_i = t_{i,n}$ Upfront funds for potential transfer to node *i*.

When the payment stops at some node *k*, node *k* includes the secret X_k in the *update_fulfill_htlc* or *update_fail_htlc* message that it sends to its upstream partner. Each node *i*, $0 < i < k$, then includes this same secret X_k in the *update_fulfill_htlc* or *update_fail_htlc* message that it sends to its upstream partner[9]. Nodes *i*-1 and *i* then update their channel state to reflect the resolution of the HTLC and the division of the staked Upfront funds. Specifically, $t_{i,k}$ Upfront funds are transferred to node *i* and the remaining $f_i - t_{i,k}$ Upfront funds are refunded to node i-1[10].

Privacy

This simple protocol allows each node to receive its correct Upfront Fee, but it reduces privacy because two routers can use the list of pairs in the *update_add_htlc* message to determine they are routing the same payment[11]. Specifically, if the payment stops at node *k*, routers *i* and *j*, where $0 < i < j \leq k$, can gain evidence that they are routing the same payment by matching:

1. the value X_k,
2. for all *m* where $j \leq m \leq n$, the values Y_m, and
3. for all *m* where $j \leq m \leq n$, the values $u_m = t_{i,m} - t_{i,m-1} = t_{j,m} - t_{j,m-1}$.

4.4 Discrete Logarithms

To prevent this loss of privacy, one must eliminate all 3 of the above techniques for matching information about a payment. The next protocol makes the following changes to achieve improved privacy:

- discrete logarithms of points (rather than hash preimages) are used as the secrets that reveal where the payment stopped,
- if a payment stops at some node *k*, each node i, $0 < i \leq k$, reveals a unique secret $p_{i,k}$ (rather than a common secret X_k) which is the discrete log of a unique point $P_{i,k}$ to its upstream partner, and

9 In both protocols presented here, a node that receives a secret from its downstream partner can instead choose to pass its own secret (which it would use if it had not received a secret from its downstream partner) to its upstream partner. However, doing so would penalize that node itself and would not penalize any other nodes.

10 In addition, both partners receive their matching funds that they had put in the burn output for the Upfront Fee.

11 This is significant if the payment uses Point Time-Locked Contracts (PTLCs), which do not use the same secret at different hops, rather than Hash Time-Locked Contracts (HTLCs), which do use the same secret at different hops.

- Upfront transfer amounts are encoded in the most significant 32 bits[12] of the discrete logs (rather than being listed explicitly in *update_add_htlc* messages).

Protocol Overview

For each i, $0 < i \leq n$, node i-1 includes the Upfront stake amount f_i and a list of points $P_{i,i}$, $P_{i,i+1}$ ... $P_{i,n}$ in the *update_add_htlc* message that it sends to node i. Nodes i-1 and i create the following contract:

- node i-1 stakes f_i Upfront funds for potential transfer to node i, and
- if node i includes a discrete log $p_{i,k}$ in the *update_fulfill_htlc* or *update_fail_htlc* message it sends to node i-1, where $p_{i,k}*G = P_{i,k}$ for some k, $i \leq k \leq n$, if $t_{i,k}$ is the value of the 32 most significant bits of $p_{i,k}$, and if $t_{i,k} \leq f_i$, then node i-1 will transfer $t_{i,k}$ Upfront funds to node i.

The onion payload for node i provides two types of secret. First, it provides the discrete log $p_{i,i}$ where $p_{i,i}*G = P_{i,i}$ and the most significant 32 bits of $p_{i,i}$ encode u_i, the Upfront transfer amount for a payment that stops at node i. Therefore, $p_{i,i}$ is the discrete log that node i gives node i-1 if the payment stops at node i.

Second, if $i < n$, the onion payload provides delta values $d_{i,k}$, $i < k \leq n$, which are the differences between the discrete log of $P_{i,k}$ and the discrete log of $P_{i+1,k}$ (where $P_{i+1,k}$ is a point in the list of points in the *update_add_htlc* message sent to node i+1). While the onion payload does not allow node i to determine the discrete log of $P_{i,k}$, it does allow node i to determine that $P_{i,k} = P_{i+1,k} + d_{i,k}*G$. Therefore, $d_{i,k}$ is the difference between the discrete log of $P_{i+1,k}$ and the discrete log of $P_{i,k}$, so if node i+1 gives node i the discrete log of $P_{i+1,k}$, node i will be able to calculate the discrete log of $P_{i,k}$. Furthermore, the most significant 32 bits of $d_{i,k}$ are a lower bound[13] on the difference between $t_{i,k}$ (the Upfront transfer to node i from node i-1) and $t_{i+1,k}$ (the Upfront transfer from node i to node i+1). Therefore, node i is able to verify that it will receive a sufficiently large net Upfront transfer by examining the most significant 32 bits of each $d_{i,k}$.

Finally, in order to reduce the size of the onion payload, rather than including the full list of delta values $d_{i,k}$, $i < k \leq n$, the onion payload includes a constant number of values that allow node i to calculate those delta values. A more detailed description of the protocol is given below.

Notation

The protocol uses discrete logarithms for the secp256k1 elliptic curve[14], where:

- ***G*** denotes the generator point and

12 Upfront transfer values will be expressed in millisatoshis, thus allowing the encoding of transfers of up to 4 million satoshis. If larger transfers are required, more than 32 bits could be used.

13 It is a lower bound because there could be a "carry-in" to the most significant 32 bits when adding $d_{i,k}$ and the discrete log of $P_{i+1,k}$.

14 Other elliptic curves could be used, as the protocol's elliptic curve calculations do not have to be performed using Bitcoin Script operators.

- ***N*** = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141 is the order of the group generated by G.

The protocol will encode Upfront transfer values as unsigned integers in the 32 most significant bits of 256-bit discrete logs, using the following notation[15].

Given an integer $m \geq 0$:

- let ***upper***(m) = *floor*($m/2^{224}$) be the result of right-shifting m 224 bit positions and
- let ***lower***(m) = (m mod 2^{224}) be the integer consisting of the 224 least significant bits in m.

Given integers $a \geq 0$ and $b \geq 0$, let ***join***(a, b) = $a*2^{224} + b$.

Note that for any integer $m \geq 0$:

- m = *join*(*upper*(m), *lower*(m)) and
- $m < (upper(m) + 1)*2^{224}$.

Note that given integers $a \geq 0$ and $b \geq 0$, $a < (upper(a) + 1)*2^{224}$ and $b < (upper(b) + 1)*2^{224}$, so $a + b < (upper(a) + upper(b) + 2)*2^{224}$ and $upper(a+b) \leq upper(a) + upper(b) + 1$.

Also, note that given integers $a \geq 0$ and $b \geq 0$, $a \geq upper(a)*2^{224}$ and $b \geq upper(b)*2^{224}$, so $a + b \geq (upper(a) + upper(b))*2^{224}$ and $upper(a+b) \geq upper(a) + upper(b)$.

Let ***max_value*** = *upper*(N) = 0xffffffff.

Note that if $upper(m) < max\_value$, then $m < N$.

Protocol

The *update_add_htlc* message sent to each node i, $0 < i \leq n$, includes:

- an Upfront stake value f_i, and
- a list of points $P_{i,n}$, $P_{i,n-1}$ ... $P_{i,i}$.

The hop payload of the onion packet that is contained within this *update_add_htlc* message includes:

- a minimum Upfront payment value u_i, and
- a value $v_{i,n}$ where $0 \leq v_{i,n} < 2^{256}$.

When it receives an *update_add_htlc* message, node i verifies that u_i is sufficiently large and that $u_i < f_i < max\_value$. Node i then calculates:

- values $v_{i,n-1}$, $v_{i,n-2}$ ... $v_{i,i}$ where $v_{i,j} = hash(v_{i,j+1})$ for $j = n-1$ ... i,

15 The notation f_i, u_i, and $t_{i,k}$ represent slightly different values in this protocol as compared to the previous protocol, due to the potential for a "carry-in" to the most significant 32 bits when adding two discrete logs.

- discrete logs $d_{i,n}$, $d_{i,n-1}$ ... $d_{i,i}$ where $d_{i,j} = join(u_i, lower(v_{i,j}))$ for $j = n$ ... i, and
- points $D_{i,n}$, $D_{i,n-1}$ ... $D_{i,i}$ where $D_{i,j} = d_{i,j}*G$ for $j = n$ ... i.

Next, node i verifies that $P_{i,i} = D_{i,i}$.

If $i < n$, node i calculates values to include in the *update_add_htlc* message sent to its downstream partner as follows:

- $f_{i+1} = f_i - u_i - 1$, and
- $P_{i+1,j} = P_{i,j} - D_{i,j}$ for $j = n$ ... i+1.

When the payment stops at node k, node k includes the discrete log $p_{k,k} = d_{k,k}$ in the *update_fulfill_htlc* or *update_fail_htlc* message that it provides to its upstream partner. Each node i, $0 < i < k$, then takes the discrete log $p_{i+1,k}$ that it received from its downstream partner and determines k[16] using the formula $p_{i+1,k}*G = P_{i+1,k}$. Next, each node i, $0 < i < k$, calculates the discrete log $p_{i,k}$ to include in the *update_fulfill_htlc* or *update_fail_htlc* message that it provides to its upstream partner using the formula $p_{i,k} = p_{i+1,k} + d_{i,k}$.

Node i-1, $0 < i \leq k$, then verifies that $P_{i,k} = p_{i,k}*G$ is one of the points that it sent to node i in the *update_add_htlc* message and that $t_{i,k} = upper(p_{i,k}) \leq f_i$. Next, nodes i-1 and i update their channel state to reflect the resolution of the HTLC and the division of the staked Upfront funds. Specifically, $t_{i,k}$ Upfront funds are transferred to node i, the remaining $f_i - t_{i,k}$ Upfront funds are refunded to node i-1, and the matching funds that each node provided for the staked Upfront funds are returned to them.

Security Analysis

Consider any node i, $0 < i \leq n$, that receives an *update_add_htlc* message for a payment and updates the current channel state to include an HTLC for that payment. There are two cases:

1. node i received an *update_fulfill_htlc* or *update_fail_htlc* message from node i+1 containing the discrete log $p_{i+1,k}$ such that:
   a) $P_{i+1,k} = p_{i+1,k}*G$ is one of the points that node i sent to node i+1 in the *update_add_htlc* message, and
   b) $t_{i+1,k} = upper(p_{i+1,k}) \leq f_{i+1}$, or
2. node i did not receive such an *update_fulfill_htlc* or *update_fail_htlc* message.

In the first case, node i calculates a transfer of $t_{i+1,k}$ Upfront funds to node i+1 and node i calculates $p_{i,k} = p_{i+1,k} + d_{i,k}$ and includes $p_{i,k}$ as the discrete log in the *update_fulfill_htlc* or *update_fail_htlc* message that it provides to its upstream partner. Note that $upper(p_{i+1,k} + d_{i,k}) \leq upper(p_{i+1,k}) + upper(d_{i,k}) + 1 \leq f_{i+1} + u_i + 1 = f_i < max\_value$ so $p_{i+1,k} + d_{i,k} < N$ and $p_{i,k} = p_{i+1,k} + d_{i,k} = ((p_{i+1,k} + d_{i,k}) \bmod N)$. Therefore, $P_{i,k} = P_{i+1,k}$

16 To be more precise, node i calculates the value n-k, as node i does not know the value of i or n.

+ $D_{i,k} = p_{i+1,k}*G + d_{i,k}*G = ((p_{i+1,k} + d_{i,k}) \bmod N)*G = ((p_{i+1,k} + d_{i,k})*G = p_{i,k}*G$, so $p_{i,k}$ is the discrete log of $P_{i,k}$. Furthermore, $t_{i,k} = upper(p_{i,k}) = upper(p_{i+1,k} + d_{i,k}) \leq upper(p_{i+1,k}) + upper(d_{i,k}) + 1 \leq f_{i+1} + u_i + 1 = f_i$, so node i met the conditions of the contract with node i-1 to receive a transfer of $t_{i,k}$ Upfront funds. Finally, note $t_{i,k} = upper(p_{i,k}) = upper(p_{i+1,k} + d_{i,k}) \geq upper(p_{i+1,k}) + upper(d_{i,k}) = t_{i+1,k} + u_i$ so node i calculates a net Upfront payment to itself of $t_{i,k} - t_{i+1,k} \geq u_i$.

In the second case, node i includes $p_{i,i} = d_{i,i}$ as the discrete log in the *update_fulfill_htlc* or *update_fail_htlc* message that it provides to its upstream partner. Note that, $p_{i,i}*G = d_{i,i}*G = D_{i,i} = P_{i,i}$ so $p_{i,i}$ is the discrete log of $P_{i,i}$. Furthermore, $t_{i,i} = upper(p_{i,i}) = upper(d_{i,i}) = u_i < f_i$, so node i met the conditions of the contract with node i-1 to receive a transfer of $t_{i,i}$ Upfront funds, so node i calculates a net Upfront payment to itself of $t_{i,i} = u_i$.

Thus, in either case node i calculates a net Upfront payment to itself of at least u_i.

Finally, consider node 0, the sender of the payment. Assume the payment stops at some node k and that each node i, $0 < i \leq k$, is a single malicious party M attempting to obtain an excessively large Upfront transfer from the sender. Note that for all j, $0 < j \leq n$, $p_{1,j} = \sum_{h=1}^{j} d_{h,j}$ and $lower(p_{1,j}) = \sum_{h=1}^{j} lower(d_{h,j}) \bmod 2^{224} = \sum_{h=1}^{j} lower(v_{h,j}) \bmod 2^{224}$. The sender selected the values $v_{i,n}$, $v_{2,n}$ ... $v_{n,n}$ randomly and independently. Therefore, in the random Oracle model, the least significant 224 bits of each discrete log $p_{1,j}$ where $0 < j \leq n$, are uniformly and independently distributed. As a result, in order to obtain an excessively large Upfront transfer from the sender, malicious party M has to calculate a discrete log of one of the points $P_{1,n}$, $P_{1,n-1}$ ... $P_{1,k+1}$ which have discrete logs with independent and uniformly random 224 least significant bits. Assuming the hardness of the discrete log problem, M will not be able to produce such a discrete log.

Privacy

Unlike the previous protocol, this protocol does not provide different routing nodes with any values that uniquely identify a given payment. Specifically, if the payment stops at node k, router i, $0 < i \leq k$, learns the value f_i, points $P_{i,n}$, $P_{i,n-1}$ ... $P_{i,i}$, the value u_i, the value $v_{i,n}$, the value $p_{i,k}$, and the value k. The values learned by two different routers i and j only allow those routers to exactly match the value n-k (which is the distance of the stopping node from the destination node). This value reduces the anonymity set but does not identify the payment.

In addition, the length of the list of points $P_{i,n}$, $P_{i,n-1}$ ... $P_{i,i}$ provides information about how close a routing node is to the destination node. This information could be reduced by artificially extending the list to include points for imaginary nodes that are past the destination. This artificial extension of the list prevents routing nodes from knowing their actual distance from the destination, unless the payment

succeeds[17]. In order to avoid revealing the distance from the destination for a successful payment, the Upfront fees for successful payments can be added to the Success fee (and the contract for the payment of a separate Upfront fee can be made contingent on the payment not being successful).

Finally, the values of the stake amounts could provide information about the path taken to the destination node. This information could be reduced by having the sender pad the Upfront fee stake amounts or using standard Upfront fee charges for all routing nodes.

Optimization

The protocol for calculating Upfront Fee stake and transfer amounts presented above can calculate an Upfront payment to node *i* that equals $u_i + 1$ (rather than u_i) due to the potential for a "carry-in" value to the upper 32 bits when calculating $p_{i,k} = p_{i+1,k} + d_{i,k}$. In such a case, the sender pays node *i* one millisatoshi more than required (as the transfer amounts are expressed in terms of millisatoshis). If this is a concern, the protocol could be modified so that the *n* most significant bits within the least significant 224 bits of discrete logs $d_{i,n}$, $d_{i,n-1}$ ... $d_{i,i}$ are set to 0 when these discrete logs are calculated.

5 Determining Hold Fee Stake And Transfer Amounts

5.1 Notation

Hold Fees are paid by nodes to all upstream nodes that are delayed beyond the payment's hold grace period. The downstream node in each channel stakes the maximum Hold Fee it could transfer by putting those funds in the burn output. If the downstream node provides an *update_fulfill_htlc* or *update_fail_htlc* message to its upstream partner after the payment's hold grace period expires, the downstream node makes a transfer of Hold Fee funds that is the product of the downstream node's *hold_usat_per_hour* value and the number of hours by which the payment was delayed, regardless of whether or not the downstream node was the cause of the delay.

We will use the following notation and terminology:

- Let *n* denote the number of payment hops, with node 0 being the sender and node *n* being the destination.
- For $0 < i \leq n$, let g_i denote the expiry of the grace period granted to node *i*.
- For $0 < i \leq n$, let h_i denote the Hold Fee funds staked by node *i*.
- For $0 < i \leq n$, let y_i denote the rate at which node *i* transfers Hold Fee funds to its upstream partner.
- For $0 < i \leq n$, let t_i denote the actual time when the payment is resolved at node *i* where:

17 In which case the success of the payment proves that it reached its destination, so the discrete log that is revealed can be understood to be provided by the destination.

- t_i is the time node i sends an *update_fulfill_htlc* or *update_fail_htlc* message to its upstream partner if the payment is resolved off-chain, or
- t_i is the expiry of the HTLC if the payment is resolved on-chain.

- For $0 < i \leq n$, let $d_i = t_i - g_i$ denote the actual delay of the payment at node i.

5.2 Protocol

The following protocol enables the correct staking and transferral of Hold Fee funds.

The *update_add_htlc* message sent to each node i, $0 < i \leq n$, includes:

- the grace period expiry g_i and
- the stake amount h_i.

If $i < n$, the hop payload of the onion packet that is contained within this *update_add_htlc* message includes:

- the outgoing grace period expiry g_{i+1} and
- the outgoing stake amount h_{i+1}.

When it receives an *update_add_htlc* message, node i, $0 < i \leq n$, calculates y_i by dividing h_i by the maximum delay possible at node i (which is the difference between the expiry of the HTLC offered to node i and the expiry of the grace period g_i). If $0 < i < n$, node i also calculates y_{i+1} and verifies that $y_{i+1} - y_i$ (which is the rate at which node i is paid a net Hold Fee for the cost of delaying its use of its capital) is sufficiently large. Node i also verifies that its hold grace period, $g_i - g_{i+1}$, is sufficiently long.

In addition, each node i calculates the amount of non-reimbursable Hold funds that it may be forced to transfer by calculating y_i times the length of time from the expiry of the HTLC with its downstream partner[18] to the expiry of the HTLC with its upstream partner. The amount of non-reimbursable Hold funds is not needed for staking and transferring Hold funds, but it is needed in order to verify that the Upfront Fee paid to node i is sufficient[19].

The Hold funds that are placed in the burn output by node i have to be divided according to how quickly the HTLC offered to node i is resolved. If that HTLC is resolved by g_i, then all of those Hold funds are refunded to node i. However, if the HTLC is resolved after g_i, node i transfers $d_i * y_i$ Hold funds to its upstream partner and refunds the remaining Hold funds to itself[20].

18 Or the length of time from g_i if $i = n$.

19 In particular, the Upfront Fee paid to node i must pay for the risk assumed by node i for transferring non-reimbursable Hold funds.

20 If both partners follow the protocol, the HTLC will be resolved at least by its expiry, in which case the value $d_i * y_i$ cannot exceed the stake amount. However, if the partners do not follow the protocol and the HTLC is resolved after its expiry, node i transfers the full stake amount h_i (and no more) from the burn output to node i-1.

5.3 Analysis

Security

The calculation of the Hold Fee transfer amount from node i to node i-1 depends only on the grace period expiry g_i, the HTLC expiry, the stake amount h_i, and the actual time t_i when the payment is resolved at node i. All of these values except t_i are known to nodes i and i-1 when they create the HTLC. The value of t_i can be calculated by both nodes when the HTLC is resolved. The risk of a disagreement in calculating t_i can be reduced by using the techniques described in Section 3 of **[Law24]**. In addition, if a Hold Fee transfer to node i-1 is required, node i can add a few seconds in calculating t_i in order to reduce the chance of a disagreement (at the expense of a minuscule increase in the Hold Fee transfer amount due to paying a Hold Fee for a few seconds longer).

Privacy

The Hold Fee stake amount does not uniquely identify a payment, but it does provide some information about the distance from the sender. The sender could pad these stake amounts in order to make it appear that the payment is farther from the sender than it really is. Such padding costs the sender am slightly increased Upfront Fee in order to cover the increased cost of capital for the seconds required for a payment that is not delayed, plus the increased risk taken on by routers for having to burn funds or having to pay for the cost of capital if they delay the payment.

5.4 Example

This subsection presents a numerical example of using Upfront, Hold and Success Fees for a 10-hop payment of 10,000 sats (satoshis), where node 0 is the payment's source and node 10 is the payment's destination. It also includes a comparison to the current protocol.

Fee-Based Spam Prevention Protocol

Assume:

- each node on the route uses the same parameters
- each routing node charges 10 msats (millisatoshis) as the fixed part of the Upfront Fee to cover the costs of calculation, communication and allocation of a slot
- each routing node charges $1 * 10^{-5}$ of the payment amount as the proportional part of the Upfront Fee to cover:
    - the cost of allocating channel funds to the payment for the seconds that can be required for a non-delayed payment, and

    - the risk of losing the payment amount (due to paying the downstream HTLC without being paid the upstream HTLC)
- each node has a *cltv_expiry_delta* of 10 hours (600 blocks)
- each node charges a Hold Fee at the rate of $2 * 10^{-5}$ / hour
    - this corresponds to a 19% (compounded) annual rate of return (as $1.00002^{24*365} = 1.19$)
- each node charges $1 * 10^{-4}$ of their nonreimbursable Hold Fee stake contribution as part of the Upfront Fee
    - this covers the risk that the node will be responsible for paying a Hold Fee due to a delay that it causes
    - the value $1 * 10^{-4}$ corresponds to causing a 10-hour delay once every 10,000 payments
    - note that a 10-hour delay is the maximum nonreimbursable delay given the *cltv_expiry_delta* value of 10 hours
    - any delay beyond 10 hours must be caused by some downstream node and thus will be paid (reimbursed) by that downstream node
- each node charges $1 * 10^{-4}$ of their Hold Fee stake contribution as part of the Upfront Fee
    - this covers the risk of burning those funds due to a partner who fails to agree to the division of the burn funds (e.g., due to a permanent failure or griefing attack)
- each node requires their partner devote (as matching funds) 1/4 of the base funds in the burn output
    - this guarantees one's partner loses at least 1/5 of what one loses in a griefing attack
    - this increases the size of the burn output by 50% (as each channel partner provides matching funds that are 1/4 of the base funds)
- each routing node charges 90 msats as the fixed part of the Success Fee
- each routing node charges $6 * 10^{-5}$ of the payment amount as the proportional part of the Success Fee

Given the above assumptions:

- each node charges a Hold Fee of $(2 * 10^{-5}$ / hour) * (10,000 sats) = $2 * 10^{-1}$ sats / hour
- each node i, $1 \leq i \leq 10$, pays a nonreimbursable Hold Fee to the i upstream nodes at the rate of i * ($2 * 10^{-1}$ sats / hour) for delays caused by node i

    - thus node *i* pays a maximum of 2**i* sats in nonreimbursed Hold Fees if node *i* causes its maximum possible 10-hour delay
- each routing node *i*, $1 \le i \le 9$, charges a Succees Fee of 90 msats + 10,000 * 6 * 10^{-5} sats = 690 msats

Given these assumptions, the following two tables calculate each node's charges and burn output contributions for Upfront and Hold Fees. An explanation of node 6's values are given after each table.

Node	Max nonreimbursable Hold Fee payment (sats)	Upfront Fee charge for nonreimbursable Hold Fee risk (msats)	Hold Fee base stake contribution (sats)	Hold Fee match stake contribution (sats)	Total Hold Fee stake contribution (sats)	Upfront Fee charge for Hold Fee burn risk (msats)
0	0	0.0	0	5	5	0.5
1	2	0.2	20	14	34	3.4
2	4	0.4	36	21	57	5.7
3	6	0.6	48	26	74	7.4
4	8	0.8	56	29	85	8.5
5	10	1.0	60	30	90	9.0
6	12	1.2	60	29	89	8.9
7	14	1.4	56	26	82	8.2
8	16	1.6	48	21	69	6.9
9	18	1.8	36	14	50	5.0
10	20	2.0	20	5	25	2.5

Table 1. Example Hold Fees and Upfront Fee charges related to Hold Fees.

For example, node 6:

- pays at most 2 sats to each of the 6 upstream nodes (nodes 0..5) for delays that node 6 causes,
- charges 12 sats * 10^{-4} = 1.2 msats in additional Upfront Fees in order to cover the risk that node 6 will have to pay a Hold Fee for a delay that it causes,
- stakes 5 * 6 * 2 = 60 sats for the maximum 2-sat Hold Fee paid by each of the 5 downstream nodes (nodes 6..10) to each of the 6 upstream nodes (nodes 0..5),
- matches 1/4 of its 60-sat Hold Fee base stake in the channel with node 5 and 1/4 of node 7's 56-sat Hold fee base stake in that channel,

- contributes 60 sats of base funds in the channel with node 5, plus it contributes 15 sats of matching funds in that channel and 14 sats of matching funds in its channel with node 7, and
- charges 89 sats * 10^{-4} = 8.9 msats for the risk that its 89-sat Hold Fee stake contribution to burn outputs will be lost due to burning those funds.

Node	Upfront Fee charge unrelated to Hold Fees (msats)	Total Upfront Fee charge (msats)	Upfront Fee base stake contribution (msats)	Upfront Fee matching stake contribution (msats)	Upfront Fee total stake contribution (msats)	Total Hold + Upfront Fee stake contribution (msats)
0	0	0.5	1,066.5	266.625	1,333.1	6,333.1
1	110	113.6	952.9	504.850	1,457.8	35,457.8
2	110	116.1	836.8	447.425	1,284.2	58,284.2
3	110	118.0	718.8	388.900	1,107.7	75,107.7
4	110	119.3	599.5	329.575	929.1	85,929.1
5	110	120.0	479.5	269.750	749.3	90,749.3
6	110	120.1	359.4	209.725	569.1	89,569.1
7	110	119.6	239.8	149.800	389.6	82,389.6
8	110	118.5	121.3	90.275	211.6	69,211.6
9	110	116.8	4.5	31.450	36.0	50,036.0
10	0	4.5	0.0	1.125	1.1	25,001.1

Table 2. Example Hold and Upfront Fee contributions.

For example, node 6:

- charges a fixed Upfront fee of 10 msats and a proportional Upfront Fee of 10,000 * 10^{-5} = 0.1 sats = 100 msats,
- charges 1.2 msats for the risk of paying a nonreimbursable Hold Fee, 8.9 msats for the risk of burning its contributions to Hold Fee stakes, and 110 msats unrelated to Hold Fees,
- stakes base funds to cover Upfront Fee transfers for payments of 119.6 + 118.5 + 116.8 + 4.5 = 359.4 msats to nodes 7..10,
- contributes 479.5 / 4 = 119.875 msats Upfront matching funds in its channel with node 5 and 359.4 / 4 = 89.85 msats Upfront matching fund in its channel with node 7,
- contributes 359.4 msats base funds and 209.725 msats matching funds for the Upfront Fee, and
- contributes 89 sats for Hold Fee stakes and 569.1 msats for Upfront Fee stakes.

The following table gives the absolute and relative sizes of the HTLC and burn outputs in each channel. An explanation of channel 5-6's values is given after the table.

Channel	HTLC output (msats)	burn output (msats)	burn output overhead (percent)
0-1	10,006,210	31,600	0.32%
1-2	10,005,520	55,429	0.55%
2-3	10,004,830	73,255	0.73%
3-4	10,004,140	85,078	0.85%
4-5	10,003,450	90,899	0.91%
5-6	10,002,760	90,719	0.91%
6-7	10,002,070	84,539	0.85%
7-8	10,001,380	72,360	0.72%
8-9	10,000,690	54,182	0.54%
9-10	10,000,000	30,007	0.30%

Table 3. Absolute and relative sizes of HTLC and burn outputs.

For example:

- the HTLC output in channel 5-6 contains the 10,000,000 msats for the payment amount and a Success Fee of 690 msats for each of the 4 downstream routing nodes (nodes 6..9),
- the burn output in channel 5-6 contains 479.5 msats in Upfront Fee base stake contributions from node 5 and 60,000 msats in Hold Fee base stake contributions from node 6 = 60,479.5 msats base funds plus an additional 60,479.5 / 4 = 15,119.875 msats of matching funds from each of nodes 5 and 6, and
- the overhead for the burn output in channel 5-6 is 90,719 / 10,002,760 = 0.91%.

Comparison With Current Protocol

Now consider four scenarios for the execution of this payment using the Fee-Based Spam Prevention protocol and the current protocol.

For the current protocol, assume:

- each routing node charges 100 msats as the fixed portion of the routing fee
- each routing node charges $7.011 * 10^{-5}$ as the proportional part of the routing fee

    - this value was chosen to match the charges for the Fee-Based Spam Prevention protocol above, minus the costs associated with losing funds by burning them (as those costs are unique to that protocol)
    - however, this value is likely too small, as it ignores the larger on-chain fees that the current protocol can have (as shown in Scenario 3 below)
- on-chain transactions cost 10 sats/vbyte (virtual byte)
- timing out an HTLC on-chain requires 388.75 vbytes
    - Commitment transaction is 168 vbytes non-witness data + 222/4 vbytes witness data
    - HTLC-timeout transaction is 94 vbytes non-witness data + 285/4 vbytes witness data
    - cost of timing out an HTLC on-chain is 10,000 msats/vbyte * 388.75 vbytes = 3,887,500 msats

Given these assumptions, with the current protocol the sender attempts to create the following HTLC outputs:

Channel	HTLC output (msats)
0-1	10,007,210
1-2	10,006,409
2-3	10,005,608
3-4	10,004,807
4-5	10,004,006
5-6	10,003,204
6-7	10,002,403
7-8	10,001,602
8-9	10,000,801
9-10	10,000,000

Table 4. Sizes of HTLC outputs when using the current Lightning protocol.

For example, the HTLC output in channel 5-6 contains the 10,000,000 msats for the payment amount and a routing fee of 100 + 10,000,000 * 7.011 * 10^{-5} = 801.1 msats for each of the 4 downstream routing nodes (nodes 6..9).

Scenario 1: Successful payment with no delay

Node	Gain using Spam Prevention protocol (msats)	Gain using current Lightning protocol (msats)
0	-10,007,276.5	-10,007,209.9
1	803.6	801.1
2	806.1	801.1
3	808.0	801.1
4	809.3	801.1
5	810.0	801.1
6	810.1	801.1
7	809.6	801.1
8	808.5	801.1
9	806.8	801.1
10	10,000,004.5	10,000,000.0

Table 5. Node outcomes for a successful payment with no delay.

For example, node 6 charges:

- an Upfront Fee of 120.1 msats and a Success Fee of 690 msats with the Spam Prevention protocol, and
- a fixed routing fee of 100 msats and a proportional routing fee of 10,000 sats * 7.011 * 10^{-5} = 701.1 msats with the current Lightning protocol.

Note that in Scenario 1 the two protocols have very similar results, with the fees for the Spam Prevention protocol being 0.9% higher overall (as 7276.5 / 7209.9 = 1.0092).

Scenario 2: Successful payment delayed one hour at the destination

Node	Gross gain (msats)	Capital costs (msats)	Net gain (msats)
0	-10,007,076.5	200.3	-10,007,276.8
1	1,003.6	200.8	802.8
2	1,006.1	201.3	804.8
3	1,008.0	201.6	806.4
4	1,009.3	201.8	807.5
5	1,010.0	201.9	808.1
6	1,010.1	201.8	808.3
7	1,009.6	201.7	807.9
8	1,008.5	201.4	807.1
9	1,006.8	201.0	805.8
10	9,998,004.5	0.5	9,998,004.0

Table 6. Node outcomes with the Spam Prevention protocol for a successful payment delayed one hour at the destination.

For example, node 6:

- charges an Upfront Fee of 120.1 msats, a Hold Fee of 10,000,000 msats * ($2 * 10^{-5}$ / hour) * 1 hour = 200 msats, and a Success Fee of 690 msats,
- has capital costs of (10,002,070 + 89,569.1) msats * ($2 * 10^{-5}$ / hour) * 1 hour = 201.8 msats, and
- receives 1,010.1 msats in fees and loses 201.8 msats in capital costs for a net gain of 808.3 msats.

Node	Gross gain (msats)	Capital costs (msats)	Net gain (msats)
0	-10,007,209.9	200.1	-10,007,410.0
1	801.1	200.1	601.0
2	801.1	200.1	601.0
3	801.1	200.1	601.0
4	801.1	200.1	601.0
5	801.1	200.1	601.0
6	801.1	200.0	601.1
7	801.1	200.0	601.1
8	801.1	200.0	601.1
9	801.1	200.0	601.1
10	10,000,000.0	0.0	10,000,000.0

Table 7. Node outcomes with the current Lightning protocol for a successful payment delayed one hour at the destination.

For example, node 6:

- charges a fixed routing fee of 100 msats and a proportional routing fee of 10,000 sats * 7.011 * 10^{-5} = 701.1 msats,
- has capital costs of 10,002,403 msats * (2 * 10^{-5} / hour) * 1 hour = 200.0 msats, and
- receives 801.1 msats in fees and loses 200.0 msats in capital costs for a net gain of 601.1 msats.

Note that in Scenario 2 the Spam Prevention protocol has a much more fair outcome, as the sender and the routing nodes are compensated for their capital costs by the node (namely the destination) that imposed those costs on them. This is an important scenario, as it is an example of a not-immediately-settled payment **[HI]**.

Scenario 3: Failed payment due to an unresponsive destination

Node	Gain using Spam Prevention protocol (msats)	Capital costs using current Lightning protocol (msats)	On-chain fees using current Lightning protocol (msats)	Net gain using current Lightning protocol (msats)
0	-1,062.0	2,001.4	0.0	-2,001.4
1	113.6	2,001.3	0.0	-2,001.3
2	116.1	2,001.1	0.0	-2,001.1
3	118.0	2,001.0	0.0	-2,001.0
4	119.3	2,000.8	0.0	-2,000.8
5	120.0	2,000.6	0.0	-2,000.6
6	120.1	2,000.5	0.0	-2,000.5
7	119.6	2,000.3	0.0	-2,000.3
8	118.5	2,000.2	0.0	-2,000.2
9	116.8	2,000.0	3,887,500.0	-3,889,500.0
10	0.0	0.0	0.0	0.0

Table 8. Node outcomes for a failed due to an unresponsive destination.

For example, node 6:

- charges an Upfront Fee of 120.1 msats,
- has capital costs of 10,002,403 msats * ($2 * 10^{-5}$ / hour) * 10 hours = 2,000.5 msats,
- is able to resolve the failed payment off-chain with nodes 5 and 7 and therefore has no on-chain costs, and
- loses 2,000.5 msats in capital costs for a net loss of 2,000.5 msats.

Note that in Scenario 3 the Spam Prevention protocol has a far better outcome than the current protocol. In particular, with the Spam Prevention protocol the sender is notified of the failed payment within seconds and no node is required to devote their capital to the payment for more than a few seconds. In contrast, with the current protocol the sender has to wait 10 hours (the time required for node 9 to time out node 10 on-chain) before discovering that the payment has failed. Furthermore, with the current protocol nodes 0 through 9 have to devote their capital to this failed payment for those 10 hours and (even more significantly) node 9 has to pay millions of msats in on-chain fees in order to resolve the payment.

The protocols behave so differently because the Spam Prevention protocol allows node 9 to fail the payment off-chain and within seconds when the destination fails to increase the burn output for the payment[21]. In contrast, the current protocol requires node 9 to go on-chain to resolve the payment.

Scenario 4: Failed payment due to insufficient funds at node 6

Node	Gain using Spam Prevention protocol (msats)	Gain using current Lightning protocol (msats)
0	-707.1	0.0
1	113.6	0.0
2	116.1	0.0
3	118.0	0.0
4	119.3	0.0
5	120.0	0.0
6	120.1	0.0
7	0.0	0.0
8	0.0	0.0
9	0.0	0.0
10	0.0	0.0

Table 9. Node outcomes for a failed payment due to insufficient funds at node 6.

For example, node 6:

- charges an Upfront Fee of 120.1 msats, and
- does not receive a routing fee when using the current Lightning protocol because the payment was unsuccessful.

Note that in Scenario 4 the Spam Prevention protocol compensates nodes 1..6 for their costs, while the current protocol fails to do so.

6 Risk Of Burning Funds

There are four ways in which a party that follows the protocol can be forced to burn funds.

First, one may have to put a Commitment transaction with a burn output on-chain in order to resolve an HTLC. In particular, this can occur when using the current Lightning channel protocol and adding burn

21 Note that this difference between the protocols did not exist in version 1.0 of this paper, as it was enabled by the bug fix in version 1.1 that separates the increase in burn funds from the creation of the HTLC output.

outputs to the Commitment transactions, as in Figure 1. However, this possibility is eliminated if the burn output is placed on a separate Payment transaction, as in Figure 2. This possibility is also eliminated when using a factory-optimized channel protocol (as in Figure 3) and/or Off-chain Payment Resolution (as in Figures 4 and 5). Having to put a Commitment transaction on-chain in order to resolve an HTLC is undesirable when using channel factories **[BDW18]** or timeout-trees **[Law23]**[22]. As a result, protocols that eliminate the risk of burning funds when resolving an HTLC may become common if channel factories or timeout-trees are used.

Second, if one's channel partner fails and cannot update the channel state for a very long time (e.g., months), one may have to close the channel unilaterally by putting their channel state transactions with a burn output on-chain. The amount of time one is willing to wait is based on the likelihood of their partner becoming responsive versus the cost of the channel's capital, and is independent of the lengths of the HTLCs' expiries.

Third, a dishonest channel partner can put channel state transactions with a burn output on-chain in order to grief their partner (while also griefing themselves). In this case, the griefer loses (at least) their matching funds, which are set high enough to discourage most or all such griefing attacks. Specifically, if each party devotes a fraction m of the amount staked for Upfront and Hold Fees as matching funds, the griefer must lose at least $m/(1+m)$ as many funds as their partner loses.

Fourth, if both parties are honest but they fail in such a way that they do not agree on the determination of Upfront and Hold Fees, they will be forced to burn those fees and their matching funds. Fortunately, there are many techniques that reduce the likelihood of such failures, including:

- adding buffers for clock skew and maximum message transmission times when determining when the downstream partner sent the payment's *update_fulfill_htlc* or *update_fail_htlc* message,
- synchronizing the channel partners' clocks by exchanging frequent time stamp messages,
- maintaining a nonvolatile log of when all *update_fulfill_htlc* or *update_fail_htlc* messages are sent or received, and
- sending multiple encrypted copies of *update_fulfill_htlc* or *update_fail_htlc* messages via independent third-parties.

If all of the multiple copies are sent but fail to reach the offerer, it is likely there was either a complete loss of communication by the offeree or a failure of the offerer. These cases can be differentiated by looking at the nonvolatile logs for the nodes' other channels. For example, if a node has 20 channels with different partners and stops receiving time-stamped messages from just one of those partners, they

22 This is because a single unresponsive party can force all or part of the factory or timeout-tree to go on-chain, thus increasing costs and decreasing scalability. Also, the time required for putting the factory or timeout-tree transactions on-chain has to be included in the *cltv_expiry_delta* parameter, thus increasing the potential for the slow resolution of Lightning payments.

can conclude that the failure was likely caused by that partner. On the other hand, if the node stops receiving messages on all 20 channels, the node can conclude that it is likely the cause of the failures (and as a result, it should agree with its partner's division of the burn output funds).

Finally, a party can lose funds if they can be psychologically manipulated to allow their partner to steal from them. For example, if Alice and Bob should each receive two thousand sats from the burn output, Bob could refuse to update the channel state unless he gets three thousand sats from the burn output (and Alice could agree in order to at least get the remaining one thousand sats). This type of bullying attack can be prevented by ensuring that all channel updates are performed automatically, rather than under human control.

7 Related Work

The protocols presented here are based on the work of many other researchers. Overviews of the spam prevention design space provided by Riard and Naumenko **[RN22]** and Shikhelman and Tikhomirov **[ST22]** helped to identify problems and possible ways to solve them.

Jager **[Jager20][Jager21]** and Teinturier **[Teint20a]** proposed using unconditional Upfront Fees and time-dependent Hold Fees to reduce spam **[Teint20b]**. The protocols presented here are based on their ideas and extend them in two ways. First, the current protocols use secrets to determine how far a payment propagated and the amount of Upfront fees to transfer within each channel. Second, the protocols given here use burn outputs and matching funds to prevent the theft of Upfront fees and encourage cooperation.

The current paper's protocols charge fees that cover all costs imposed, which is an idea originally proposed by Towns **[Towns23]**. These protocols differ from Towns's proposal by not requiring the closure of a channel when the downstream partner is unresponsive. This difference could be important in practice, as an unresponsive partner is likely the most common cause of payment delay. The observation that the Hold Fee can be calculated independently by both channel partners also comes from Towns **[Towns23]**.

The idea of using a Commitment transaction burn output for fees in order to prevent fee theft and to encourage cooperation between partners comes from Riard **[Riard22]** who wrote:

> On the structure of the monetary strategy, I think there could be a solution to implement a proof-of-burn, where the fee is captured in a commitment output sending to a provably unspendable output. Theoretically, it's interesting as "unburning" the fee is dependent on counterparty cooperation, the one potentially encumbering the jamming risk.

In addition, Riard and Naumenko mention the potential for collateral burning and "leave finding a trust-minimized scheme for collateral burning for future work" **[RN22]**. The current paper can be viewed as an example of such future work.

Finally, other researchers have investigated reputation-based and hybrid fee-and-reputation-based approaches to spam mitigation. While some of these have shown real potential to mitigate spam **[KS24]**, there are several advantages to using pure fee-based protocols such as those presented here. First, even if a reputation-based protocol has not been attacked successfully, it is very difficult to prove that the protocol can never be attacked successfully. In contrast, fee-based protocols can be shown to force attackers to pay for all significant costs that they impose on others. Second, reputation-based protocols often put participants into discrete categories, thus allowing attackers to push the boundaries of acceptable behavior without being penalized. Third, it can take time and expense to build a good reputation, thus penalizing those who are new to Lightning. Fourth, fee-based protocols go beyond penalizing spam by forcing all parties to pay for all significant costs that they impose on others. As a result, fee-based protocols create a more efficient market for Lightning that should lead to reduced costs and/or improved service.

8 Conclusions

This paper presents protocols for assigning and collecting fees for Lightning services. These protocols force the parties to pay for all significant costs that they impose on others and the fees they charge cannot be stolen by self-interested parties. It is hoped these protocols will be used to both reduce spam and improve the efficiency of the Lightning Network.

9 Acknowledgments

The author would like to thank David A. Harding for finding a bug (related to the collection of Hold Fees) in the original version of this paper and for identifying the increase in payment latency that results from the most straightforward fix for that bug. The author would also like to thank Clara Shikhelman for her suggestion to include numerical examples of the protocol.

References

BDW18 Burchert, Decker and Wattenhofer, "Scalable Funding of Bitcoin Micropayment Channel Networks", http://dx.doi.org/10.1098/rsos.180089

BOLT2 BOLT 02 - Peer Protocol, https://github.com/lightning/bolts/blob/master/02-peer-protocol.md

Hard25 Harding, "Re: Fee-Based Spam Prevention For Lightning", https://delvingbitcoin.org/t/fee-based-spam-prevention-for-lightning/1524/2

HI Hold Invoices, https://bitcoinops.org/en/topics/hold-invoices/

Jager20 Jager, "Hold fees: 402 Payment Required for Lightning itself", https://www.mail-archive.com/lightning-dev@lists.linuxfoundation.org/msg02080.html

Jager21 Jager, "Hold fee rates as DoS protection (channel spamming and jamming)", https://www.mail-archive.com/lightning-dev@lists.linuxfoundation.org/msg02212.html

KS24 Kirk-Cohen and Shikhelman, "Hybrid Jamming Mitigation: Results and Updates", https://delvingbitcoin.org/t/hybrid-jamming-mitigation-results-and-updates/1147

Law22 Law, "Factory Optimized Protocols For Lightning", https://github.com/JohnLaw2/ln-factory-optimized

Law23 Law, "Scaling Lightning With Simple Covenants", https://github.com/JohnLaw2/ln-scaling-covenants

Law24 Law, "A Fast, Scalable Protocol For Resolving Lightning Payments", https://github.com/JohnLaw2/ln-opr

Riard22 Riard, "Unjamming lightning (new research paper)", https://www.mail-archive.com/lightning-dev@lists.linuxfoundation.org/msg02996.html

RN22 Riard and Naumenko, "Lightning Jamming Book", https://jamming-dev.github.io/book/

ST22 Shikhelman and Tikhomirov, "Unjamming Lightning", https://github.com/s-tikhomirov/ln-jamming-simulator/blob/master/unjamming-lightning.pdf

Teint20a Teinturier, "Re: Hold fees: 402 Payment Required for Lightning itself", https://www.mail-archive.com/lightning-dev@lists.linuxfoundation.org/msg02106.html

Teint20b Teinturier, "Spam Prevention", https://github.com/t-bast/lightning-docs/blob/master/spam-prevention.md

Towns23 Towns, "Re: LN Summit 2023 Notes", https://www.mail-archive.com/lightning-dev@lists.linuxfoundation.org/msg03267.html

Appendix A: Detailed Protocol Specifications

This appendix gives detailed protocols for determining and collecting Upfront and Hold Fees. In the following consider a payment with n hops, where node 0 is the sender and node n is the destination[23].

A.1 Changes To Existing Parameters

In order to simplify the protocols, all times (including HTLC and grace period expiries) will be expressed in terms of wall clock time[24], rather than block height[25], as follows:

23 If desired for privacy, the sender can determine Hold Fees and Upfront Fees for an extended route that is padded at one or both ends to include nodes that are not part of the actual payment route.

24 Relative times will be expressed in milliseconds and absolute times will be expressed as milliseconds since the Unix epoch.

25 Alternatively, grace periods could be expressed as wall clock times and HTLC expiries could continue to be expressed as block heights. In this case, provision needs to be made for either determining Hold Fee delays in terms of block heights or determining Hold Fee delays in terms of wall clock time and handling cases where the calculated Hold Fee transfer amount exceeds the amount staked (due to the HTLC's expiry occurring later than expected).

- *cltv_expiry_delta_msec_i* field replaces the *cltv_expiry_delta_i* field in the *channel_update* message sent by node *i*,
- *min_final_cltv_expiry_msec_i* field replaces the *min_final_cltv_expiry_i* field in the payment invoice provided by node *i*,
- *cltv_expiry_msec_i* field replaces the *cltv_expiry_i* field in the *update_add_htlc* message sent to node *i*, and
- *outgoing_cltv_expiry_msec_i* field replaces the *outgoing_cltv_value_i* field in the onion per-hop for node *i*.

A.2 Upfront Fees

This subsection presents a detailed protocol for calculating and collecting Upfront Fees using discrete logarithms.

Message Additions

In order to support Upfront Fees, the following fields are added to the following messages:

- *channel_update* messages sent by node *i* add:
  - *upfront_charge_base_msat_i* (fixed part of Upfront Fee charge covers: 1) the computation and communication costs of routing the payment and 2) the cost of allocating a Commitment transaction output to the payment)
  - *upfront_charge_proportional_millionths_i* (proportional part of Upfront Fee charge covers: 1) cost of capital for grace period, 2) risk of losing capital due to failing to satisfy HTLC to upstream partner despite downstream partner satisfying HTLC, and 3) risk of burning Upfront funds staked in downstream channel)
  - *upfront_charge_hold_nonreimbursable_millionths_i* (charge for risk of having to pay Hold Fee due to causing a delay)
  - *upfront_charge_hold_stake_millionths_i* (charge for risk of burning Hold funds staked in upstream channel)
  - *partner_burn_match_thousandths_i* (fraction of this channel's Upfront and Hold Fee stake amounts that this node's partner must contribute to the burn output)
- *update_add_htlc* messages received by node *i* add:
  - *upfront_stake_msat_i* (f_i = amount upstream node in this channel stakes for Upfront Fees for given payment)

- u*pfront_point_list_i* ($P_{i,n}$, $P_{i,n-1}$ ... $P_{i,i}$ = list of Upfront points for which node *i* must provide a single discrete log in order to determine an Upfront transfer amount to node *i*)

- onion per-hop payload for node *i* adds:
  - u*pfront_fee_msat_i* (u_i = minimum Upfront Fee paid to node *i* for given payment)
  - *upfront_base_value_i* ($v_{i,n}$ = value from which hashed values $v_{i,n-1}$, $v_{i,n-2}$ ... $v_{i,i}$ are calculated)
- *update_fulfill_htlc* and *update_fail_htlc* messages sent by node *i* add:
  - *upfront_discrete_log_i* ($p_{i,k}$ = discrete log of Upfront point $P_{i,k}$ where $t_{i,k} = upper(p_{i,k})$ is the amount of Upfront funds to transfer to node *i* for a payment that stopped at node *k*)

Sender Protocol Additions

In order to support Upfront Fees the sender of a Lightning payment performs the following additional actions:

- For each node *i* along the payment route, $0 < i \le n$, the sender calculates:
  - $u_i = upfront\_fee\_msat\_i \ge upfront\_charge\_base\_msat\_i + upfront\_charge\_proportional\_millionths\_i * amount\_msat\_i / 1{,}000{,}000 + upfront\_charge\_hold\_nonreimbursable\_millionths\_i * hold\_nonreimbursable\_msat\_i$[26] $/ 1{,}000{,}000 + upfront\_charge\_hold\_stake\_millionths\_i * hold\_stake\_msat\_i$[27] $/ 1{,}000{,}000$.
- For each node *i* along the payment route, $0 < i \le n$, the sender calculates:
  - $f_i = upfront\_stake\_msat\_i$ = sum of ($upfront\_fee\_msat\_j + 1$) for all j, $i \le j \le n$.
  - a random $v_{i,n}$ where $0 \le v_{i,n} < 2^{256}$,
  - values $v_{i,n-1}$, $v_{i,n-2}$ ... $v_{i,i}$ where $v_{i,j} = hash(v_{i,j+1})$ for $j = n\text{-}1 \ldots i$,
  - discrete logs $d_{i,n}$, $d_{i,n-1}$ ... $d_{i,i}$ where $d_{i,j} = join(u_i, lower(v_{i,j}))$ for $j = n \ldots i$,
  - points $D_{i,n}$, $D_{i,n-1}$ ... $D_{i,i}$ where $D_{i,j} = d_{i,j} * G$ for $j = n \ldots i$, and
  - points $P_{i,i}$, $P_{i-1,i}$ ... $P_{1,i}$ where $P_{h,i} = \sum_{j=h}^{i} D_{j,i}$.

Router And Receiver Protocol Additions

In order to support Upfront Fees, each node *i* along the payment route, $0 < i \le n$, performs the following additional actions:

26 The rules for calculating *hold_nonreimbursable_msat_i* are given in Appendix A.3.
27 The rules for calculating *hold_stake_msat_i* are given in Appendix A.3.

- When it receives an *update_add_htlc* message, node i:
  - verifies that u_i = *upfront_fee_msat_i* ≥ *upfront_charge_base_msat_i* + *upfront_charge_proportional_millionths_i* * *amount_msat_i* / 1,000,000 + *upfront_charge_hold_nonreimbursable_millionths_i* * *hold_nonreimbursable_msat_i* / 1,000,000 + *upfront_charge_hold_stake_millionths_i* * *hold_stake_msat_i* / 1,000,000,
  - verifies that u_i = *upfront_fee_msat_i* < f_i < *max_value*,
  - calculates values $v_{i,n-1}$, $v_{i,n-2}$ ... $v_{i,i}$ where $v_{i,j} = hash(v_{i,j+1})$ for $j = n-1 \dots i$,
  - calculates discrete logs $d_{i,n}$, $d_{i,n-1}$ ... $d_{i,i}$ where $d_{i,j} = join(u_i, lower(v_{i,j}))$ for $j = n \dots i$,
  - calculates points $D_{i,n}$, $D_{i,n-1}$ ... $D_{i,i}$ where $D_{i,j} = d_{i,j}*G$ for $j = n \dots i$,
  - verifies that $P_{i,i} = D_{i,i}$, and if $i < n$:
    - calculates $f_{i+1} = f_i - u_i - 1$, and
    - $P_{i+1,j} = P_{i,j} - D_{i,j}$ for $j = n \dots i+1$.
- When upstream node i sends an *update_add_htlc* message to downstream node $i+1$, both nodes:
  - stake Upfront funds by moving f_{i+1} from node i's output to the burn output in the channel's current transactions,
  - provide node i's matching funds by moving *partner_burn_match_thousandths_*($i+1$) * f_{i+1} from node i's output to the burn output in the channel's current transactions, and
  - provide node $i+1$'s matching funds by moving *partner_burn_match_thousandths_i* * f_{i+1} from node $i+1$'s output to the burn output in the channel's current transactions.
- If node i receives an *update_fulfill_htlc* or *update_fail_htlc* message with *upfront_discrete_log_*($i+1$) = $p_{i+1,k}$ field, node i:
  - calculates $t_{i+1,k} = upper(p_{i+1,k})$,
  - verifies that $t_{i+1,k} \leq f_{i+1}$,
  - calculates the index k such that $p_{i+1,k}*G = P_{i+1,k}$,
  - calculates $p_{i,k} = p_{i+1,k} + d_{i,k}$, and
  - sends an *update_fulfill_htlc* or *update_fail_htlc* message with *upfront_discrete_log_i* = $p_{i,k}$ field to its upstream partner.
- If node i does not receive an *update_fulfill_htlc* or *update_fail_htlc* message with *upfront_discrete_log_*($i+1$) = $p_{i+1,k}$ field that meets the above conditions, node i:

    - calculates the discrete log $p_{i,i} = d_{i,i}$ and
    - sends an *update_fulfill_htlc* or *update_fail_htlc* message with *upfront_discrete_log_i* = $p_{i,i}$ field to its upstream partner.
- When downstream node i+1 sends an *update_fulfill_htlc* or *update_fail_htlc* message to upstream node i with an *upfront_discrete_log_*(i+1) = $p_{i+1,k}$ field where $p_{i+1,k}*G = P_{i+1,k}$ and $t_{i+1,k}$ = *upper*($p_{i+1,k}$) ≤ f_{i+1}, both nodes:
    - transfer $t_{i+1,k}$ funds from the burn output to node i+1's output in the channel's current transactions,
    - refund f_{i+1} - $t_{i+1,k}$ funds from the burn output to node i's output in the channel's current transactions,
    - refund node i's matching funds by moving *partner_burn_match_thousandths_*(i+1) * f_{i+1} from the burn output to node i's output in the channel's current transactions, and
    - refund node i+1's matching funds by moving *partner_burn_match_thousandths_i* * f_{i+1} from the burn output to node i+1's output in the channel's current transactions.

A.3 Hold Fees

Message Changes

In order to support the calculation and low-latency collection of Hold Fees, the following changes are made to the following messages:

- payment invoices add:
    - *min_final_hold_grace_period_delta_msec* (minimum grace period required at the destination)
- *funding_locked* messages add:
    - *per_commitment_point*_2 (the third per-commitment point which follows *per_commitment_point*_1 contained in the same *funding_locked* message and *per_commitment_point*_0 contained in an *open_channel* or *accept_channel* message)
- *revoke_and_ack_x* messages contain:
    - *per_commitment_secret_x* (as in the current *revoke_and_ack_x* message)
    - *per_commitment_point_*(x+3) (rather than *per_commitment_point_*(x+2) as in the current *revoke_and_ack_x* message)
- *channel_update* messages sent by node i add:

    - *hold_grace_period_delta_msec_i* (minimum grace period required by this node)
    - *hold_charge_billionths_per_hour_i* (charge for cost of holding this node's capital past the payment's grace period)
    - *partner_burn_match_thousandths_i* (fraction of Upfront and Hold Fee stake amounts that one's partner must contribute to the burn output)
- *update_add_htlc* messages received by node *i* add:
    - *hold_grace_period_expiry_msec_i* (expiry of this payment's grace period at node *i*)
    - *hold_stake_msat_i* (h_i = amount downstream node in this channel stakes for Hold Fees for given payment)
- If the current channel is not the payment's last channel, the onion per-hop payload received by node *i* adds:
    - *outgoing_hold_grace_period_expiry_msec_i* (expiry of this payment's grace period at node *i*+1)
    - *outgoing_hold_stake_msat_i* (h_{i+1} = amount downstream node in next channel stakes for Hold Fees for given payment)
- *update_fulfill_htlc* and *update_fail_htlc* messages sent by node *i* add:
    - *timestamp_msec_i* (time when message was sent)

In addition, an *update_remove_htlc* message is added where:

- the *update_remove_htlc* message contains:
    - *id* (identifier of the HTLC that is being removed)

Sender Protocol Additions

In order to support Hold Fees, the sender of a Lightning payment performs the following additional actions:

- The sender calculates *hold_charge_msat_per_hour*_0 = *amount_msat*_1 * *hold_charge_billionths_per_hour*_0 / 1,000,000,000 (rate at which sender charges a Hold Fee)
- For each node *i* along the payment route, $0 < i \leq n$, the sender calculates:

- *hold_grace_period_expiry_msec_i = current_time_msec*[28] *+ buffer_msec*[29] *+ min_final_hold_grace_period_delta_msec_n* + sum of *hold_grace_period_delta_msec_j* for all *j*, $i \leq j < n$,
- *hold_exposure_msec_i = cltv_expiry_msec_i - hold_grace_period_expiry_msec_i* (time node *i* can have to pay Hold Fees, regardless of whether or not they are reimbursable)
- if $i < n$, *hold_nonreimbursable_msec_i = cltv_expiry_msec_i - cltv_expiry_msec_*(*i*+1) (time node *i* can have to pay Hold Fees while it cannot be reimbursed by its downstream partner)
- if $i = n$, *hold_nonreimbursable_msec_n = cltv_expiry_msec_n - hold_grace_period_expiry_msec_n* (time node *n* can have to pay Hold Fees)
- *hold_transfer_msat_per_hour_i* = sum of *hold_charge_msat_per_hour_j* for all *j*, $0 \leq j < i$ (rate at which node *i* transfers Hold Fee funds to its upstream partner)
- *hold_stake_msat_i* = h_i = *hold_transfer_msat_per_hour_i * hold_exposure_msec_i* / 3,600,000 (amount node *i* stakes for Hold Fee transfers)
- *hold_nonreimbursable_msat_i*[30] *= hold_transfer_msat_per_hour_i * hold_nonreimbursable_msec_i* / 3,600,000 (non-reimbursable amount node *i* may have to transfer for Hold Fees)
- *hold_charge_msat_per_hour_i* = (*amount_msat_*(*i*+1) + h_i) * *hold_charge_billionths_per_hour_i* / 1,000,000,000 (rate at which node *i* charges a Hold Fee)

Router And Receiver Protocol Additions

In order to support Hold Fees, each node along the payment route performs the following additional actions:

- When it receives an *update_add_htlc* message, node *n*:
  - verifies that *hold_grace_period_expiry_msec_n* is at least *min_final_hold_grace_period_delta_msec_n* milliseconds in the future,
  - calculates *hold_nonreimbursable_msec_n = cltv_expiry_msec_n - hold_grace_period_expiry_msec_n*,

28 *current_time_msec* is the wall clock time when the sender starts performing calculations for this payment.
29 *buffer_msec* is selected by the sender in order to cover the time the sender spends creating the payment message and the time required to propagate the payment request to the destination node.
30 *hold_nonreimbursable_msat_i* is not required for calculating Hold Fees, but it is used in calculating Upfront Fees as described in Appendix A.2.

- calculates *hold_transfer_msat_per_hour_n = hold_stake_msat_n* * 3,600,000 / (*cltv_expiry_msec_n - hold_grace_period_expiry_msec_n*), and
- calculates *hold_nonreimbursable_msat_n = hold_stake_msat_n*.

- When it receives an *update_add_htlc* message, node *i*, $0 < i < n$,
  - verifies that *hold_grace_period_expiry_msec_i - outgoing_hold_grace_period_expiry_msec_i* ≥ *hold_grace_period_delta_msec_i*,
  - calculates *hold_exposure_msec_i = cltv_expiry_msec_i - hold_grace_period_expiry_msec_i*,
  - calculates *hold_nonreimbursable_msec_i = cltv_expiry_msec_i - outgoing_cltv_expiry_msec_i*,
  - calculates *hold_transfer_msat_per_hour_i = hold_stake_msat_i* * 3,600,000 / (*cltv_expiry_msec_i - hold_grace_period_expiry_msec_i*),
  - calculates *hold_transfer_msat_per_hour_*(*i*+1) = *outgoing_hold_stake_msat_i* * 3,600,000 / (*outgoing_cltv_expiry_msec_i - outgoing_hold_grace_period_expiry_msec_i*),
  - verifies *hold_transfer_msat_per_hour_*(*i*+1) - *hold_transfer_msat_per_hour_i* ≥ *hold_charge_billionths_per_hour_i * amount_msat_i* / 1,000,000,000, and
  - calculates *hold_nonreimbursable_msat_i = hold_transfer_msat_per_hour_i * hold_nonreimbursable_msec_i* / 3,600,000.
- When upstream node *i* sends an *update_add_htlc* message to downstream node *i*+1, both nodes[31]:
  - stake Hold funds by moving h_{i+1} from node *i*+1's output to the burn output in the channel's current transactions,
  - provide node *i*'s matching funds by moving *partner_burn_match_thousandths_*(*i*+1) * h_{i+1} from node *i*'s output to the burn output in the channel's current transactions, and
  - provide node *i*+1's matching funds by moving *partner_burn_match_thousandths_i* * h_{i+1} from node *i*+1's output to the burn output in the channel's current transactions.
- When nodes *i* and *i*+1 resolve a payment, both nodes:
  - calculate *t_*(*i*+1) = *timestamp_msec_*(*i*+1) if the payment was resolved off-chain,
  - calculate *t_*(*i*+1) = *cltv_expiry_msec_*(*i*+1) if the payment was resolved on-chain,

31 The staging of these updates is defined in the "Message Transfers And Staging Of Updates" subsection below.

- calculate *hold_delay_hour_*(*i*+1) = max(0, *t_*(*i*+1) - *hold_grace_period_expiry_msec_*(*i*+1)) / 3,600,000,
- calculate *hold_transfer_msat_*(*i*+1) = min(h_{i+1}, *hold_delay_hour_*(*i*+1) * *hold_transfer_msat_per_hour_*(*i*+1)),
- transfer *hold_transfer_msat_*(*i*+1) funds from the burn output to node *i*'s output in the channel's current transactions,
- refund h_{i+1} - *hold_transfer_msat_*(*i*+1) funds from the burn output to node *i*+1's output in the channel's current transactions,
- refund node *i*'s matching funds by moving *partner_burn_match_thousandths_*(*i*+1) * h_{i+1} from the burn output to node *i*'s output in the channel's current transactions, and
- refund node *i*+1's matching funds by moving *partner_burn_match_thousandths_i* * h_{i+1} from the burn output to node *i*+1's output in the channel's current transactions.

Message Transfers And Staging Of Updates

The following definitions will be used is this subsection:

- a commitment transaction is *current* if it is signed and has not been revoked and
- a commitment transaction is *usable* unless:
  - it includes an HTLC output for an HTLC offered by the party holding the transaction and
  - the party holding the transaction has sent an *update_remove_htlc* message for that HTLC.

When a new channel is created, the channel partners send a *funding_created* message and a *funding_signed* message in order to give each party their first signed commitment transaction (also called the refund transaction) for the channel. Then, each node generally alternates between receiving a *commitment_signed* message and sending a *revoke_and_ack* message.

However, a node receives 2 consecutive *commitment_signed* messages (without sending an intervening *revoke_and_ack* message) whenever the first *commitment_signed* message gives the node exactly 2 current commitment transactions, the more recent of which includes an increased burn amount (but no HTLC output) for an HTLC offered by the node.

Similarly, a node sends 2 consecutive *revoke_and_ack* messages (without receiving an intervening *commitment_signed* message) whenever the node has exactly 3 current commitment transactions, the most recent of which is usable.

These rules eliminate the possibility of a race between an incoming *commitment_signed* message and an outgoing *revoke_and_ack* message, thus allowing each party to know their partner's number of current commitment transactions. The implications of these rules are given in the Analysis subsection.

In the flows presented below, parties are generally able to send additional messages between those sent in successive steps. However, when one party sends two consecutive *commitment_signed* messages (as in steps **a4** and **a5** or steps **b4** and **b5** below) that party cannot send any other messages between the two consecutive ones specified[32].

To simplify the presentation, the flows below focus on a single HTLC that is being offered. If multiple HTLCs are being offered in the same step (**a2**, **b2**, **c2** or **d2**) and if the *revoke_and_ack* message that commits to the increased burn amount (in step **a3**, **b3**, **c3** or **d3**) is received after the grace period of **any** of those offered HTLCs, then **all** of those offered HTLCs must be removed with an *update_remove_htlc* message (step **b6** or **d5**).

Four separate flows are possible for offering a new HTLC in a channel.

First, if upstream node Alice has exactly 1 current commitment transaction when she offers an HTLC to Bob, and Bob commits to the increased burn amount for the HTLC within the grace period[33], the flow is:

- **a1:** *update_add_htlc* message is sent to Bob
  - increased burn amount is pending for Bob
  - Bob has at least 1 current commitment transaction
    - increased burn amount is not in Bob's latest current commitment transaction
- **a2:** *commitment_signed* message is sent to Bob
  - ... and the increased burn amount is in Bob's latest commitment transaction
  - Bob has at least 2 current commitment transactions
    - increased burn amount is not in Bob's earliest current commitment transaction
    - increased burn amount is in Bob's latest commitment transaction
- **a3:** *revoke_and_ack* message is sent to Alice and is received during the grace period
  - ... and all of Bob's earlier commitment transactions have been revoked, and the increased burn amount is pending for Alice

32 This constraint is not strictly necessary, but it is included in order to simplify the specification.

33 More precisely, this flow occurs if 1) Alice receives the *revoke_and_ack* message in Step **a3** during the grace period of **all** of the HTLCs for which the burn amount was increased in step **a2**, and 2) Alice has exactly 1 current commitment transaction immediately before she receives the *commitment_signed_i* message in Step **a4**.

    - Bob has at least 1 current commitment transaction
        - increased burn amount is in Bob's earliest current commitment transaction
- **a4:** *commitment_signed_i* message is sent to Alice
    - ... and the increased burn amount is in Alice's latest commitment transaction, and the HTLC output is pending for Alice
    - Alice has exactly 2 current commitment transactions: A_{i-1} and A_i
        - increased burn amount is not in A_{i-1}
        - increased burn amount is in A_i
- **a5:** *commitment_signed_*(*i*+1) message is the next message sent to Alice
    - ... and the increased burn amount and HTLC output are in Alice's latest commitment transaction
    - Alice has exactly 3 current commitment transactions: A_{i-1}, A_i and A_{i+1}
        - increased burn amount is not in A_{i-1}
        - increased burn amount is in A_i, HTLC output is not in A_i
        - increased burn amount and HTLC output are in A_{i+1}
- **a6:** *revoke_and_ack_*(*i*-1) message is sent to Bob
    - ... and Alice's earliest current commitment transaction has been revoked
    - Alice has exactly 2 current commitment transaction: A_i and A_{i+1}
        - increased burn amount is in A_i, HTLC output is not in A_i
        - increased burn amount and HTLC output are in A_{i+1}
- **a7:** *revoke_and_ack_i* message is sent to Bob
    - ... and Alice's previous commitment transaction has been revoked, and the HTLC output is pending for Bob
    - Alice has exactly 1 current commitment transaction: A_{i+1}
        - increased burn amount and HTLC output are in A_{i+1}
- **a8:** *commitment_signed* message is sent to Bob
    - ... and the HTLC output is in Bob's latest commitment transaction

    - Bob has at least 2 current commitment transactions
        - increased burn amount is in Bob's earliest current commitment transaction, HTLC output is not in Bob's earliest current commitment transaction
        - increased burn amount and HTLC output are in Bob's latest commitment transaction
- **a9:** *revoke_and_ack* message is sent to Alice
    - ... and all of Bob's earlier commitment transactions have been revoked
    - Bob has at least 1 current commitment transaction
        - increased burn amount and HTLC output are in Bob's earliest current commitment transaction

On the other hand, if upstream node Alice has exactly 1 current commitment transaction when she offers an HTLC to Bob, and Bob commits to the increased burn amount for the HTLC after the grace period[34], the flow is:

- **b1:** *update_add_htlc* message is sent to Bob
    - increased burn amount is pending for Bob
    - Bob has at least 1 current commitment transaction
        - increased burn amount is not in Bob's latest current commitment transactions
- **b2:** *commitment_signed* message is sent to Bob
    - ... and the increased burn amount is in Bob's latest commitment transaction
    - Bob has at least 2 current commitment transactions
        - increased burn amount is not in Bob's earliest current commitment transaction
        - increased burn amount is in Bob's latest commitment transaction
- **b3:** *revoke_and_ack* message is sent to Alice and is received after the grace period
    - ... and all of Bob's earlier commitment transactions have been revoked, and the increased burn amount is pending for Alice
    - Bob has at least 1 current commitment transaction
        - increased burn amount is in Bob's earliest current commitment transaction

34 More precisely, this flow occurs if 1) Alice receives the *revoke_and_ack* message in Step **b3** after the grace period of **any** of the HTLCs for which the burn amount was increased in step **b2**, and 2) Alice has exactly 1 current commitment transaction immediately before she receives the *commitment_signed_i* message in Step **b4**.

- **b4:** *commitment_signed_i* message is sent to Alice
  - ... and the increased burn amount is in Alice's latest commitment transaction, and the HTLC output is pending for Alice
  - Alice has exactly 2 current commitment transactions: A_{i-1} and A_i
    - increased burn amount is not in A_{i-1}
    - increased burn amount is in A_i
- **b5:** *commitment_signed_*(i+1) message is the next message sent to Alice
  - ... and the increased burn amount and HTLC output are in Alice's latest commitment transaction
  - Alice has exactly 3 current commitment transactions: A_{i-1}, A_i and A_{i+1}
    - increased burn amount is not in A_{i-1}
    - increased burn amount is in A_i, HTLC output is not in A_i
    - increased burn amount and HTLC output are in A_{i+1}
- **b6:** *update_remove_htlc* message is sent to Bob
  - ... and the failed HTLC (without increased burn amount or HTLC output) is pending for Bob
- **b7:** *revoke_and_ack_*(i-1) message is sent to Bob
  - ... and Alice's earliest current commitment transaction has been revoked
  - Alice has exactly 2 current commitment transaction: A_i and A_{i+1}
    - increased burn amount is in A_i, HTLC output is not in A_i
    - increased burn amount and HTLC output are in A_{i+1}
- **b8:** *commitment_signed* message is sent to Bob
  - ... and the failed HTLC (without increased burn amount or HTLC output) is in Bob's latest commitment transaction
  - Bob has at least 2 current commitment transactions
    - increased burn amount is in Bob's earliest current commitment transaction, HTLC output is not in Bob's earliest current commitment transaction

        - failed HTLC (without increased burn amount or HTLC output) is in Bob's latest commitment transaction
- **b9:** *revoke_and_ack* message is sent to Alice
    - ... and all of Bob's earlier commitment transactions have been revoked, and the failed HTLC (without increased burn amount or HTLC output) is pending for Alice
    - Bob has at least 1 current commitment transaction
        - failed HTLC (without increased burn amount or HTLC output) is in Bob's earliest current commitment transaction
- **b10:** *commitment_signed_*(i+2) message is sent to Alice
    - ... and the failed HTLC (without increased burn amount or HTLC output) is in Alice's latest commitment transaction
    - Alice has exactly 3 current commitment transaction: A_i, A_{i+1} and A_{i+2}
        - increased burn amount is in A_i, HTLC output is not in A_i
        - increased burn amount and HTLC output are in A_{i+1}
        - failed HTLC (without increased burn amount or HTLC output) is in A_{i+2}
- **b11:** *revoke_and_ack_i* message is sent to Bob
    - ... and Alice's earliest current commitment transaction has been revoked
    - Alice has exactly 2 current commitment transactions: A_{i+1} and A_{i+2}
        - increased burn amount and HTLC output are in A_{i+1}
        - failed HTLC (without increased burn amount or HTLC output) is in A_{i+2}
- **b12:** *revoke_and_ack_*(i+1) message is sent to Bob
    - ... and Alice's previous commitment transaction has been revoked
    - Alice has exactly 1 current commitment transaction: A_{i+2}
        - failed HTLC (without increased burn amount or HTLC output) is in A_{i+2}

Next, if Alice has exactly 2 current commitment transaction when she offers an HTLC to Bob, and Bob commits to the increased burn amount for the HTLC within the grace period[35], the flow is:

35 More precisely, this flow occurs if 1) Alice receives the *revoke_and_ack* message in Step **c3** during the grace period of **all** of the HTLCs for which the burn amount was increased in step **c2**, and 2) Alice has exactly 2 current commitment transactions immediately before she receives the *commitment_signed_i* message in Step **c4**.

- **c1:** *update_add_htlc* message is sent to Bob
  - increased burn amount pending for Bob
  - Bob has at least 1 current commitment transaction
    - increased burn amount is not in Bob's latest current commitment transaction
- **c2:** *commitment_signed* message is sent to Bob
  - ... and the increased burn amount is in Bob's latest commitment transaction
  - Bob has at least 2 current commitment transactions
    - increased burn amount is not in Bob's earliest current commitment transaction
    - increased burn amount is in Bob's latest commitment transaction
- **c3:** *revoke_and_ack* message is sent to Alice and is received during the grace period
  - ... and all of Bob's earlier commitment transactions have been revoked, and the increased burn amount is pending for Alice
  - Bob has at least 1 current commitment transaction
    - increased burn amount is in Bob's earliest current commitment transaction
- **c4:** *commitment_signed_i* message is sent to Alice
  - ... and the increased burn amount is in Alice's latest commitment transaction
  - Alice has exactly 3 current commitment transactions: A_{i-2}, A_{i-1} and A_i
    - increased burn amount is not in A_{i-2}
    - increased burn amount is not in A_{i-1}
    - increased burn amount is in A_i
- **c5:** *revoke_and_ack_*(*i*-2) message is sent to Bob
  - ... and Alice's earliest current commitment transaction has been revoked
  - Alice has exactly 2 current commitment transaction: A_{i-1} and A_i
    - increased burn amount is not in A_{i-1}
    - increased burn amount is in A_i
- **c6:** *revoke_and_ack_*(*i*-1) message is sent to Bob

    - ... and Alice's previous commitment transaction has been revoked, and the HTLC output is pending for Bob
    - Alice has exactly 1 current commitment transaction: A_i
        - increased burn amount is in A_i
- **c7:** *commitment_signed* message is sent to Bob
    - ... and the HTLC output is in Bob's latest commitment transaction
    - Bob has at least 2 current commitment transactions
        - increased burn amount is in Bob's earliest current commitment transaction, HTLC output is not in Bob's earliest current commitment transaction
        - increased burn amount and HTLC output are in Bob's latest commitment transaction
- **c8:** *revoke_and_ack* message is sent to Alice
    - ... and all of Bob's earlier commitment transactions have been revoked, and the HTLC output is pending for Alice
    - Bob has at least 1 current commitment transaction
        - increased burn amount and HTLC output are in Bob's earliest current commitment transaction
- **c9:** *commitment_signed_*(i+1) message is sent to Alice
    - ... and the increased burn amount and HTLC output are in Alice's latest commitment transaction
    - Alice has exactly 2 current commitment transactions: A_i and A_{i+1}
        - increased burn amount is in A_i
        - increased burn amount and HTLC output are in A_{i+1}
- **c10:** *revoke_and_ack_i* message is sent to Bob
    - ... and Alice's earlier current commitment transaction has been revoked
    - Alice has exactly 1 current commitment transaction: A_{i+1}
        - increased burn amount and HTLC output are in A_{i+1}

Finally, if Alice has exactly 2 current commitment transactions when she offers an HTLC to Bob, and Bob commits to the increased burn amount for the HTLC after the grace period[36], the flow is:

- **d1:** *update_add_htlc* message is sent to Bob
  - increased burn amount is pending for Bob
  - Bob has at least 1 current commitment transaction
    - increased burn amount is not in Bob's latest current commitment transactions
- **d2:** *commitment_signed* message is sent to Bob
  - ... and the increased burn amount is in Bob's latest commitment transaction
  - Bob has at least 2 current commitment transactions
    - increased burn amount is not in Bob's earliest current commitment transaction
    - increased burn amount is in Bob's latest commitment transaction
- **d3:** *revoke_and_ack* message is sent to Alice and is received after the grace period
  - ... and all of Bob's earlier commitment transactions have been revoked, and the increased burn amount is pending for Alice
  - Bob has at least 1 current commitment transaction
    - increased burn amount is in Bob's earliest current commitment transaction
- **d4:** *commitment_signed_i* message is sent to Alice
  - ... and the increased burn amount is in Alice's latest commitment transaction
  - Alice has exactly 3 current commitment transactions: A_{i-2}, A_{i-1} and A_i
    - increased burn amount is not in A_{i-2}
    - increased burn amount is not in A_{i-1}
    - increased burn amount is in A_i
- **d5:** *update_remove_htlc* message is sent to Bob
  - ... and the failed HTLC (without increased burn amount or HTLC output) is pending for Bob
- **d6:** *revoke_and_ack_*(*i*-2) message is sent to Bob

36 More precisely, this flow occurs if 1) Alice receives the *revoke_and_ack* message in Step **d3** after the grace period of **any** of the HTLCs for which the burn amount was increased in step **d2**, and 2) Alice has exactly 2 current commitment transactions immediately before she receives the *commitment_signed_i* message in Step **d4**.

    - ... and Alice's earliest current commitment transaction has been revoked
    - Alice has exactly 2 current commitment transaction: A_{i-1} and A_i
        - increased burn amount is not in A_{i-1}
        - increased burn amount is in A_i
- **d7:** *revoke_and_ack*_(i-1) message is sent to Bob
    - ... and Alice's previous commitment transaction has been revoked
    - Alice has exactly 1 current commitment transaction: A_i
        - increased burn amount is in A_i
- **d8:** *commitment_signed* message is sent to Bob
    - ... and the failed HTLC (without increased burn amount or HTLC output) is in Bob's latest commitment transaction
    - Bob has at least 2 current commitment transactions
        - increased burn amount is in Bob's earliest current commitment transaction, HTLC output is not in Bob's earliest current commitment transaction
        - failed HTLC (without increased burn amount or HTLC output) is in Bob's latest commitment transaction
- **d9:** *revoke_and_ack* message is sent to Alice
    - ... and all of Bob's earlier commitment transactions have been revoked, and the failed HTLC (without increased burn amount or HTLC output) is pending for Alice
    - Bob has at least 1 current commitment transaction
        - failed HTLC (without increased burn amount or HTLC output) is in Bob's earliest current commitment transaction
- **d10:** *commitment_signed*_(i+1) message is sent to Alice
    - ... and the failed HTLC (without increased burn amount or HTLC output) is in Alice's latest commitment transaction
    - Alice has exactly 2 current commitment transaction: A_i and A_{i+1}
        - increased burn amount is in A_i
        - failed HTLC (without increased burn amount or HTLC output) is in A_{i+1}

- **d11:** *revoke_and_ack_i* message is sent to Bob
  - ... and Alice's earliest current commitment transaction has been revoked
  - Alice has at least 1 current commitment transactions: A_{i+1}
    - failed HTLC (without increased burn amount or HTLC output) is in A_{i+1}

Analysis

The rules on the order in which a node receives *commitment_signed* messages and sends *revoke_and_ack* messages constrain how the number and type of current commitment transactions can evolve. Starting with a single usable current commitment transaction, 3 cases are possible. These cases are illustrated in the following tables, where an 'X' denotes that a given commitment transaction is current at a given stage, a black 'X' denotes a usable commitment transaction, and a red 'X' denotes an unusable one.

Case 1: Gain and loss of 1 current commitment transaction.

In this case, a node starts at a given stage S with a single usable current commitment transaction T. It receives 1 *commitment_signed* message and sends 1 *revoke_and_ack* message. This case is illustrated in Table 10.

	Stage S	**Stage S+1**	**Stage S+2**
commitment transaction T	**X**	**X**	
commitment transaction T+1		**X**	**X**

Table 10. Gain and loss of 1 current commitment transaction.

Case 2: Gain and loss of 2 current commitment transactions.

In this case, a node receives 2 consecutive *commitment_signed* messages and sends 2 consecutive *revoke_and_ack* messages. This case is illustrated in Table 11.

	Stage S	**Stage S+1**	**Stage S+2**	**Stage S+3**	**Stage S+4**
commitment transaction T	**X**	**X**	**X**		
commitment transaction T+1		**X**	**X**	**X**	
commitment transaction T+2			**X**	**X**	**X**

Table 11. Gain and loss of 2 current commitment transactions.

Case 3: Gain and loss of 3 current commitment transactions.

In this case, a node receives 2 consecutive *commitment_signed* messages and sends 1 *revoke_and_ack* message, receives 1 *commitment_signed* message, and sends 2 consecutive *revoke_and_ack* messages. This case is illustrated in Table 12.

	Stage S	**Stage S+1**	**Stage S+2**	**Stage S+3**	**Stage S+4**	**Stage S+5**	**Stage S+6**
commitment transaction T	**X**	**X**	**X**				
commitment transaction T+1		**X**	**X**	**X**	**X**		
commitment transaction T+2			**X**	**X**	**X**	**X**	
commitment transaction T+3					**X**	**X**	**X**

Table 12. Gain and loss of 3 current commitment transactions.

Case 1 occurs when commitment transaction T+1 **does not** include an increased burn amount (but no HTLC output) for a new HTLC offered by the given node. Cases 2 and 3 occur when commitment transaction T+1 **does** include an increased burn amount (but no HTLC output) for a new HTLC offered by the given node. Cases 2 and 3 differ based on whether commitment transaction T+2 (which includes an increased burn amount and HTLC output for the new HTLC) is usable or not.

Case 2 corresponds to the first flow (steps **a1** through **a9**) given in the previous subsection and Case 3 corresponds to the second flow (steps **b1** through **b12**) given in the previous subsection[37]. The third flow (steps **c1** through **c10**) and fourth flow (**d1** through **d11**) occur when commitment transaction T+3 in Table 12 includes an increased burn amount (but not HTLC output) for the given HTLC[38].

37 In both of these flows, commitment transaction A_i corresponds to commitment transaction T+1 in Table 11.
38 In both of these flows, commitment transaction A_i corresponds to commitment transaction T+3 in Table 12.