A Fast, Scalable Protocol For Resolving Lightning Payments

John Law

October 25, 2024

Version 1.0

Abstract

This paper presents a new protocol for resolving Lightning payments, called the Off-chain Payment Resolution (OPR) protocol, which allows Lightning payments to be resolved without putting any transactions related to the payment on-chain, even in the worst case. As a result, it solves three important problems: 1) support for arbitrarily small payments, 2) resolution of all Lightning payments within seconds, and 3) greatly improved scalability. Furthermore, it supports payments to and from casual users without requiring a watchtower service (or high availability).

The OPR protocol is a *griefer-penalized* protocol, as a party can be made to lose certain funds by a channel partner who griefs them. However, 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

The security of all known Lightning payment protocols breaks down when the incremental cost of resolving a payment on-chain exceeds the value of the payment.

For example, if Alice uses the current Lightning protocol to offer an HTLC to Bob, and Bob fulfills the HTLC by providing the required hash preimage before its expiry, Alice is supposed to update the channel state off-chain to reflect the payment of the HTLC to Bob. However, if Alice fails to do so, Bob's only recourse is to put an HTLC-success transaction on-chain. If the cost of putting the HTLC-success transaction on-chain exceeds the value of the payment, resolving the HTLC on-chain is more costly for Bob than allowing Alice to claim the payment. Therefore, Alice is incentivized to violate the protocol by not giving the payment's funds to Bob off-chain, as she will receive those funds unless Bob acts against his immediate self-interest.

This paper presents a new Lightning payment resolution protocol, called the Off-chain Payment Resolution (OPR) protocol, that has the following properties:

1. all payments are resolved correctly, provided both parties are self-interested and able to implement the protocol, regardless of how small the payment is,
2. all payments are resolved within seconds (as compared to hours for the current protocol),
3. there is never a need for a party to go on-chain to resolve a payment (thus greatly improving scalability). and
4. casual users can send and receive payments without having a watchtower service (and without maintaining high availability).

As a result, the OPR protocol appears to be well-suited to supporting small, everyday Lightning payments. In addition, the OPR protocol can be used for large payments, either directly or by dividing the large payment into many smaller ones.

The OPR protocol has weaker security than a trust-free protocol (in which one cannot lose funds if they follow the protocol), but stronger than a trust-based protocol (in which one's funds can be stolen). The OPR protocol is a *griefer-penalized* protocol, as an honest party can be made to lose certain funds by a dishonest channel partner who griefs them. However, the griefer also has to lose a comparable amount of funds. As a result, an honest party who only creates channels with self-interested parties will never be subject to a griefing attack.

2 Security Of The Current Lightning Protocol

As noted above, with the current Lightning protocol an offeree of an HTLC may be forced to put an HTLC-success transaction on-chain. The HTLC-success transaction requires about 702 vbytes **[BOLT3]**, and the additional input in the transaction that spends the HTLC-sucess transaction's output requires about 316 vbytes, for a total of about 1018 incremental vbytes to claim a payment on-chain. Assuming $50k per BTC and the historically common fee rates of 10 sat/vbyte to 100 sat/vbyte **[TF]**, the incremental cost for claiming a payment on-chain is in the range of $5 to $50.

Of course, even if the incremental cost of claiming a payment exceeds the value of the payment, that does not mean that all Lightning nodes will try to steal the payment. For example, the expected value of future routing fees could exceed the value of the currently-outstanding small payments, thus preventing thefts. Also, there are significant software engineering barriers to stealing small payments, as the offeree of an HTLC **will** choose to lose funds by securing a small payment on-chain if the offeree is running standard software that implements the Lightning protocol.

On the other hand, there are situations where the current Lightning protocol could lead to an attempt to steal funds. For example, if Alice wants to close a channel with Bob, she could create one large payment and very many small payments for which Alice offers Bob HTLCs. Next, Alice stops responding to Bob's messages after receiving the HTLCs' hash preimages. As a result, Bob will put his Commitment transaction and the large payment's HTLC-success transaction on-chain. In addition, Bob must either lose funds by claiming all of the small payments on-chain, or forfeit those payments to Alice.

This attack is costless for Alice, has at least some possibility of allowing her to steal funds, appears to be due to an involuntary failure (rather than malicious behavior), and is performed when she no longer needs to maintain her reputation with Bob in order to obtain future routing fees.

3 Off-chain Payment Resolution (OPR) Protocol

The OPR protocol operates in conjunction with a channel protocol that creates a series of signed off-chain transactions reflecting the current channel state. Old channel states can be replaced by a new channel state via a revocation key (as in the current Lightning protocol **[BOLT2]**) or a per-commitment key (as in the Tunable Penalty **[Law22b]** or Fully-Factory-Optimized **[Law22c]** protocol). The details of how to maintain the channel state are presented in Section 9.

When the channel is created, each party gets signed transactions with three outputs, one of which pays to each party, with the third one being a *burn output*. The burn output pays to anyone after a 20-year delay (so the burn output will be claimed by a miner 20 years in the future). Each party must contribute a fixed amount (called *base funds*) to the burn output.

In order to route a new payment, an HTLC is created for the amount of the payment plus routing fees. Each party gets new channel state transactions where the value of the HTLC is moved from the offerer's output to the burn output. In addition, each party adds a fixed fraction of the value of the HTLC (called *matching funds*) to the burn output.

As in the current Lightning protocol, the HTLC pays to the offeree only if the offeree provides the offerer with a hash preimage before the HTLC's expiry. However, the expiry is set at most seconds in the future, based on the parties' *htlc_expiry_delta_msec* and *min_final_expiry_delta_msec* parameters. These parameters are set to provide enough time to resolve the HTLC off-chain by receiving an *update_fulfill_htlc* or *update_fail_htlc* message, signing updated channel state transactions, and propagating the payment resolution to the upstream channel. These parameters also include buffers for communication and computation delays caused by unusually heavy traffic, but they **do not** include time for putting channel state transactions on-chain.

If an HTLC is resolved successfully before its expiry, the HTLC's funds are moved to the offeree's output and both parties' matching funds for the HTLC are returned to them. If an HTLC fails, the HTLC's funds are moved to the offerer's output and both parties' matching funds for the HTLC are returned to them.

As long as both parties agree on whether the HTLC succeeded or failed, they will agree on the new channel state transactions that resolve the HTLC and they will both get back their matching funds for the HTLC. Finally, if both parties agree on the resolution of all HTLCs and they want to close the channel, they can create a Cooperative Close transaction that returns their base funds to them and eliminates the burn output.

4 Burned Funds

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

First, 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 base and 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 HTLC amount as matching funds, the griefer must lose at least $m/(1+m)$ as many funds as their partner loses (and at least their base funds).

Second, if an honest party's channel partner completely fails and cannot update the channel state for a very long time (e.g., months), the honest party can be forced to close the channel unilaterally by putting their channel state transactions with a burn output on-chain. The amount of time an honest party 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, if both parties are honest but they fail in such a way that they do not agree on the resolution of an HTLC, they will be forced to burn the HTLC's funds and their base and matching funds. Fortunately, there are many techniques that reduce the likelihood of such failures.

In order to determine if an HTLC was resolved successfully, a node has to determine if the required hash preimage was provided before the HTLC's expiry. All hash preimage messages can include a time stamp recording when they were sent, and each node can keep a time-stamped nonvolatile log of each hash preimage that it sends or receives. This log can be used to determine the result of an HTLC, even if the node crashes when the HTLC was being resolved. Channel partners can keep their clocks synchronized by exchanging frequent time stamp messages, and the *htlc_expiry_delta_msec* parameters can include a buffer for clock skew.

The greatest challenge in getting agreement on whether or not an HTLC was resolved successfully occurs when the offeree sends the hash preimage before the expiry, but the offerer does not receive the preimage until after the expiry. This case can be made less frequent by calculating the maximum communication latency *L* between channel partners and never sending a hash preimage less than *L* before the HTLC's expiry,

Also, multiple encrypted copies of hash preimage messages can be sent using different communication paths. If all of these multiple copies are sent but fail to reach the offerer before the HTLC's expiry, 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 can conclude that the failure was likely caused by that partner. On the other hand, if the node stops receiving time-stamped messages on all 20 channels, the node can conclude that it is likely the cause of the failures.

5 HTLC Failures

In addition to the risk of burning funds, a routing node can lose funds if it causes an HTLC to fail.

If a node failure prevents it from fulfilling an upstream HTLC, the node will lose the value of the HTLC because it is still liable for paying the corresponding downstream HTLC. The risk of losing HTLC funds by having to pay a downstream HTLC without receiving an upstream HTLC exists in the current Lightning protocol. However, the OPR protocol greatly increases this risk because it commits the node to delaying the resolution of the HTLC by at most seconds, as opposed to hours in the current Lightning protocol.

This risk is covered by Lightning routing fees, so it is worth quantifying the risk in order to determine whether or not those fees will be prohibitive. If we assume the average incremental latency required to resolve an HTLC is 100 milliseconds and the average payment traverses 11 hops, each HTLC will be resolved an average of 600 milliseconds after it is created. If each node fails (due to a crash, delay, or protocol error) randomly 10 times a day, thus causing it to lose the value of all unresolved HTLCs, that node will lose the value of one out of every 14,400 HTLCs. Therefore, increasing the routing fee by 1/14,400 = 0.007% of the HTLC value per node (and thus 0.077% per payment) will cover the cost of node failures that cause HTLC failures.

As a result, it seems plausible that the OPR protocol could be used to implement fast off-chain payments without imposing excessive routing fees.

6 Bullying

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 million sats from the burn output, Bob could refuse to update the channel state unless he gets three million sats from the burn output (and Alice could agree in order to at least get the remaining one million sats).

This type of bullying attack can be prevented by ensuring that all channel updates are performed automatically, rather than under human control.

7 Scalability

The OPR protocol has remarkable scaling properties.

First, the addition of an HTLC to a channel does not add any outputs to the channel state transactions. As a result, even if the channel state is put on-chain, there is no incremental on-chain footprint due to the HTLC. In addition, there is no protocol-defined upper bound on the number of active HTLCs per channel.

Second, the OPR protocol's resolution of an HTLC never requires any transactions to be put on-chain. This is extremely important for scalability, as channel factories **[BDW18][DRO18][Law22d]** and timeout-trees **[Law21][Law23b]** have been proposed for opening and closing channels off-chain, and hierarchical channels **[Law23a]** have been proposed for resizing channels off-chain. As a result, without the OPR protocol, the on-chain resolution of HTLCs is likely to be the greatest limitation to Lightning's scalability.

8 Usability

The OPR protocol's guaranteed resolution of a payment attempt within seconds makes it much more attractive to casual users than the current Lightning protocol, which could require waiting hours to find out that a payment attempt failed. The fact that the payment's receiver never has to go on-chain to resolve the payment means that the OPR protocol supports one-shot receives (that is, receives without having to be online at some specific time after the receive operation is started) **[Law22a]**. Asynchronous trampoline payments (as described in Section 3.6 of **[Law22a]**) can be used for payments between casual users that are not online at the same time.

The OPR protocol, combined with the use of revocation keys or per-commitment keys (as described in Section 9) to maintain the current channel state, also allows casual users to send and receive bitcoin without using a watchtower service. This is accomplished by having the casual user pair with a dedicated user to create an unannounced payment channel that is not used to route payments for others. The casual user sets their *to_self_delay* parameter to 3 months plus 1 day[1] and pays their channel partner a fee for their partner's cost of capital (due to the casual user's long *to_self_delay* parameter). The casual user must check the blockchain at least once every 3 months, and if they detect that their partner has attempted to put an old channel state on-chain, they must use their revocation private key or per-commitment private key to prevent the theft of their funds.

A casual user who sends a payment is likely to be online for the seconds that it takes to resolve the payment attempt. However, if the casual user becomes unavailable during those seconds, they should not assume that the payment failed simply because they did not receive the payment's receipt (hash preimage) prior to the expiry of their channel's HTLC. Instead, the casual user should reestablish time-stamped communication with their partner (ideally via multiple communication paths) after the expiry of the payment's HTLC in order to determine the payment's resolution. As a result, the casual user will be able to obtain a receipt (hash preimage) or notification of the payment's failure within seconds, while minimizing the likelihood of a disagreement with their partner that would cause the payment's funds to be burned.

1 The additional day allows for the casual user to revoke transactions for an old state that were put on-chain by their partner.

9 Maintaining The Channel State

As noted above, either revocation keys or per-commitment keys can be used to maintain the current channel state.

9.1 Revocation Keys

In the current Lightning protocol, old channel states are made unusable via revocation keys. In order to revoke Alice's transactions for channel state *i*, Alice gives Bob her private key for her per-commitment public key *i*. Bob creates a linear combination of this private key and his revocation basepoint private key to create his revocation private key *i*. Knowledge of Bob's revocation private key *i* can be used to spend all outputs from Alice's transactions for state *i*. As a result, if Alice puts her transactions for state *i* on-chain, Bob will be able to claim all of the channel's funds.

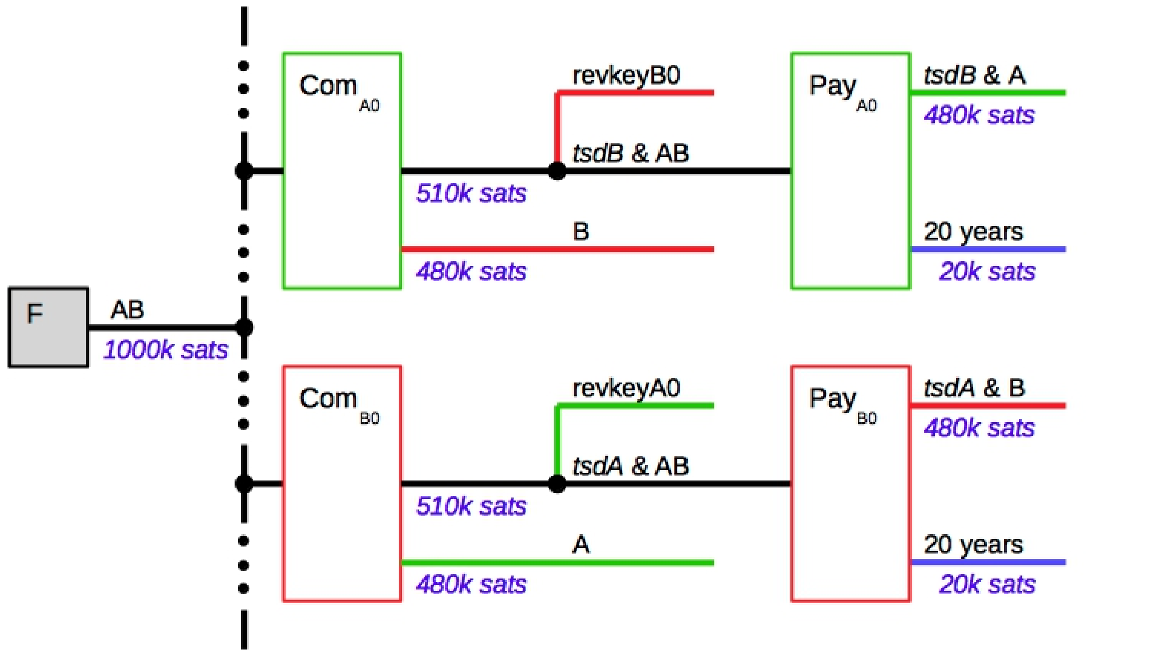
The simplest way to modify the current Lightning protocol to implement the OPR protocol consists of adding a burn output (that pays to miners 20 years in the future) to Alice's state *i* transactions. However, this approach is not safe, as Alice could still put an old state *i* on-chain, in which case the funds in the burn output for state *i* would be unavailable to Alice and Bob. Because there is no guaranteed relationship between the size of the burn output in an old state *i* and Bob's current funds in the channel, there is no guarantee as to the relative size of Alice's and Bob's losses if Alice griefs Bob by putting state *i* on-chain.

This problem could be addressed by allowing the burn output in state *i* to also be spent using revocation private key *i*. This change assures that Alice cannot grief Bob by putting an old state on-chain without Bob gaining all of the funds in the channel (minus fees). Unfortunately, it introduces another problem, as Alice could put the current state on-chain. If Alice does that, neither Alice nor Bob could spend the burn output. However, they have 20 years in which to decide that they would each be better off if they split the burn output, so they may choose to mutually sign a transaction that does so. The problem with this outcome is that it eliminates or reduces Alice's penalty for putting the current state on-chain, so she may choose to do so in anticipation of such an outcome.

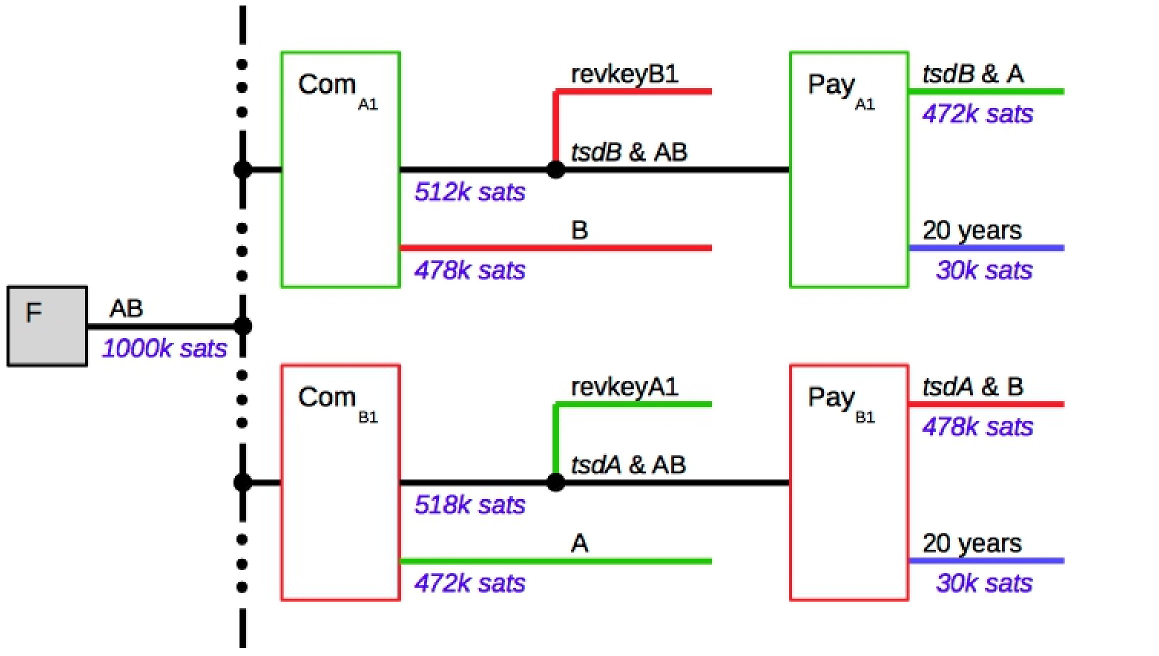
This problem is directly analogous to the bullying problem discussed in Section 6, and once again the solution is to guarantee that all channel updates are performed automatically, rather than under human control. One way to do this is to create a new transaction that has both the current state's *to_self* and burn outputs, but which can only be put on-chain after the given state is proven to not have been revoked. This solution is shown in Figures 1 and 2 below.

In Figures 1 and 2, the on-chain Funding transaction (F) funds a channel with 1000k sats. Alice and Bob create and sign off-chain Commitment (Com) and Payment (Pay) transactions with the given inputs (on the left side) and outputs (on the right side). The conditions required to spend an output are shown above the output and the values of the output is given below the output.

The capital letter A (B) indicates that Alice's (Bob's) private key is required to spend the output, and revkeyAi indicates that Alice's (Bob's) revocation private key *i* is required to spend the output. Outputs labeled with *tsdA* (*tsdB*) can only be spent after a relative delay of Alice's (Bob's) *to_self_delay* parameter. Outputs labeled 20 years can be spent by anyone after a relative delay of 20 years. Outputs that branch can be spent by different transactions that meet the given branch's requirements.

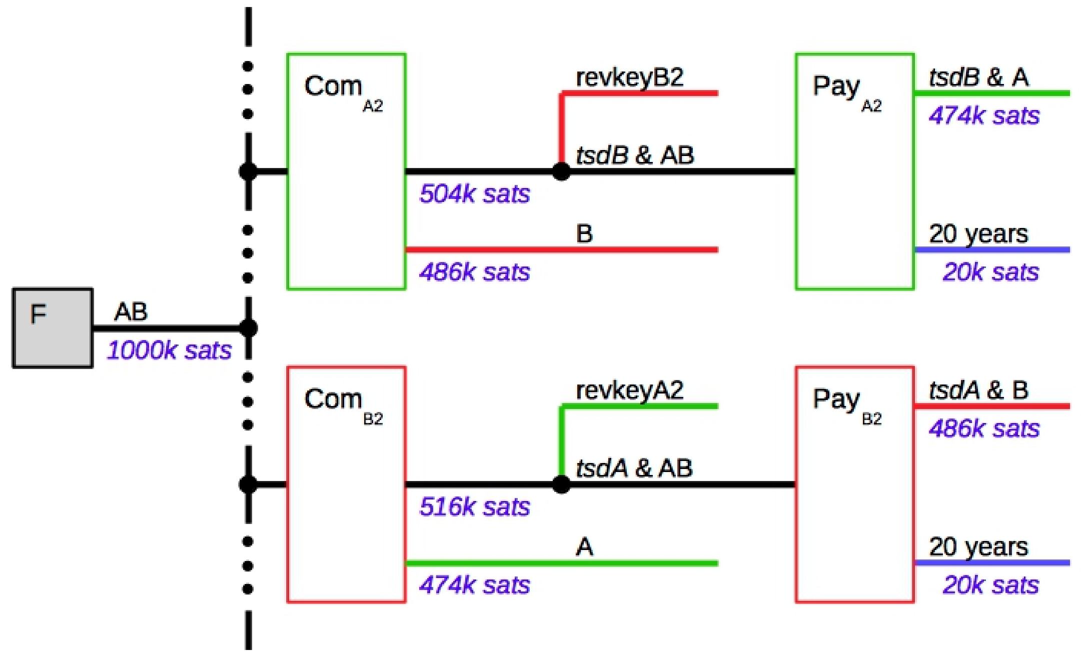


a) Initial channel state with 20k sat base funds from each party (split between fees and burn output)

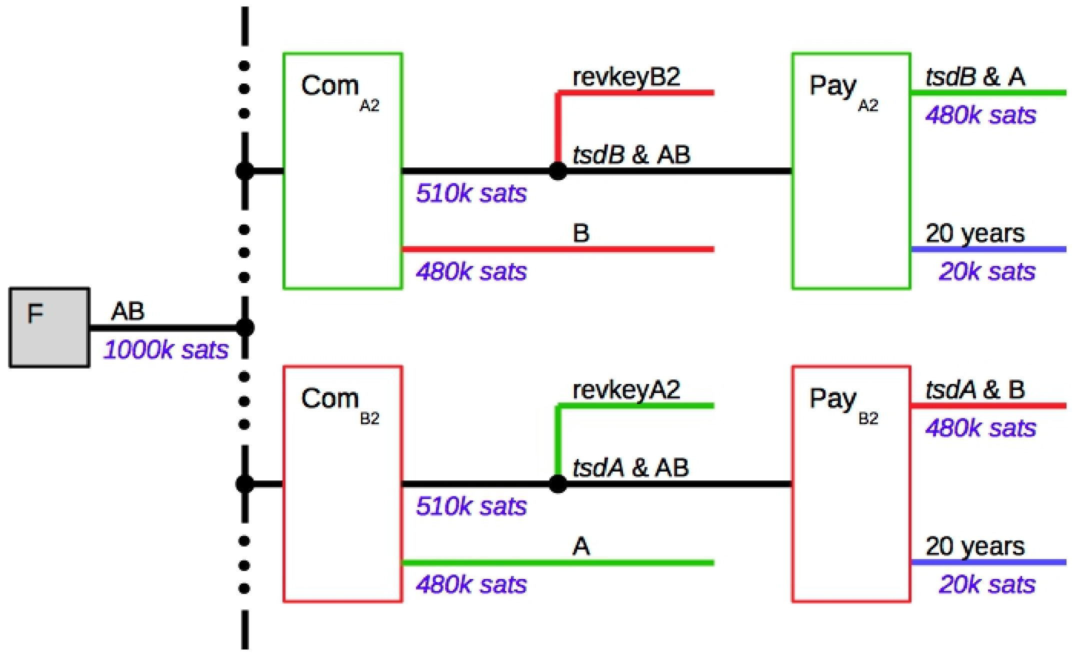


b) Channel state #1 with 6k sat HTLC from A to B (plus 2k sat matching funds from each party)

Figure 1. Use of revocation keys to implement (a) an initial channel state and (b) an HTLC worth 6k sats offered by Alice to Bob (plus 2k sat matching funds from each party).



a) Channel state #2 when 6k sat HTLC from A to B succeeds



b) Channel state #2 when 6k sat HTLC from A to B fails

Figure 2. Resolution of the HTLC from Figure 1 when it (a) succeeds or (b) fails.

In the description of the OPR protocol given in Section 3, each party's base funds are moved to the burn output. However, the fees required for putting a transaction on-chain act very much like the funds in the burn output, so an optimization is to allow base funds to be used for transaction fees. In Figure 1, each party contributes 20k sats in base funds, half of which go to paying transaction fees and half of which go to the burn output.

9.2 Per-Commitment Keys

The Lightning protocol uses a revocation private key to revoke transactions for old channel states. The revocation private key cannot be known to the party whose transactions are being revoked, in order to prevent that party from "revoking" them and claiming their outputs for themselves.

In contrast, the Tunable Penalty **[Law22b]** and Fully-Factory-Optimized **[Law22c]** protocols use a per-commitment private key to revoke transactions for old channel states, where the party that put the old transactions on-chain **does** know the per-commitment private key that revokes them. This is possible because when these protocols revoke transactions for an old channel state, the party that revokes the transactions does not gain any of the funds in the channel. Instead, the revoking party obtains a penalty payment, where the size of the penalty is independent of the funds in the channel.

In both of those protocols, each party has an on-chain Individual transaction with a penalty-valued[2] output. In order to put channel state *i* on-chain, a party first puts their State transaction for state *i* on-chain that spends their Individual transaction's output. This State transaction has a single penalty-valued output that can be spent with this party's per-commitment private key *i*, thus gaining the penalty value. Alternatively, after waiting a relative delay given by the other party's *to_self_delay* parameter, the party that put the State transaction on-chain can spend its output with their state *i* Commitment transaction. This Commitment transaction has another input that spends the channel's Funding transaction (which can be off-chain in the Fully-Factory-Optimized protocol).

These protocols can be modified to implement the OPR protocol by using a Commitment transaction that has three outputs, one for each party plus a burn output. The resulting protocol is shown in Figure 3 below. In that figure, pckeyAi (pckeyBi) is Alice's (Bob's) per-commitment private key for state *i*.

2 Plus fees for putting the associated State transaction on-chain.

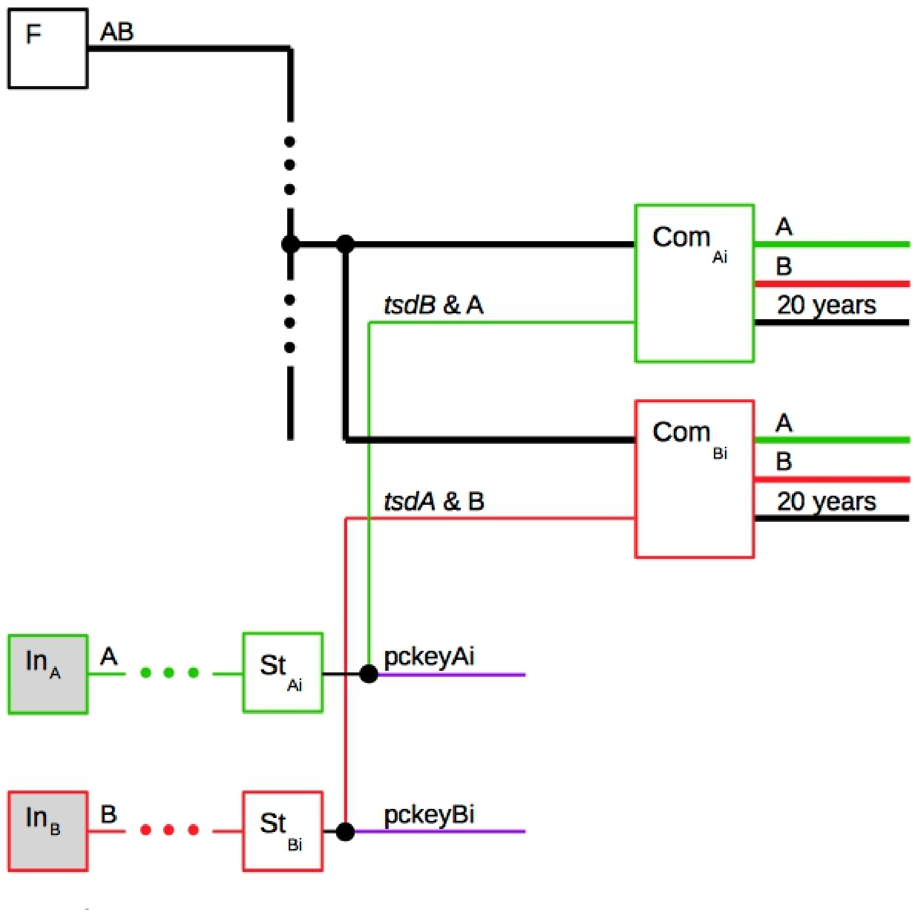


Figure 3. Implementation of the OPR protocol using per-commitment keys to invalidate old states.

The Tunable Penalty and Fully-Factory-Optimized protocols differ only in the transactions they put on-chain in order to resolve HTLCs. Because no transactions are put on-chain to resolve HTLCs with the OPR protocol, the Tunable Penalty and Fully-Factory-Optimized protocols are unified when they are used to implement the OPR protocol. The result is a simplified protocol that gets all of the features of the Tunable Penalty, Fully-Factory-Optimized and OPR protocols. Furthermore, as was noted in Section 8, the OPR protocol lets casual users perform watchtower-free sends and receives, thus providing all of the usability properties of the Watchtower-Free **[Law22a]** and Fully-Factory-Optimized-Watchtower-Free **[Law22c]** protocols. These results are summarized below.

Using per-commitment keys to maintain the channel state with the OPR protocol (as shown in Figure 3) and supporting casual users (as described in Section 8) results in the following properties:

1. erroneously attempting to put an old channel state on-chain results in paying a penalty, the value of which is tunable and independent of the channel's value **[Law22b]**,

2. dedicated users can use a watchtower service that only requires storage that is logarithmic (as opposed to linear) in the number of old channel states (due to the use of per-commitment keys as described in Section 4 of **[Law22b]**),
3. if the channel is created in a channel factory or timeout-tree, the maximum latency of the payment is independent of the time required to put any channel factory or timeout-tree transactions on-chain **[Law22c]**,
4. there is no need to put channel factory or timeout-tree transactions on-chain to resolve a payment **[Law22c]**,
5. casual users can receive payments in a one-shot manner **[Law22a]**,
6. casual users can send and receive payments without using a watchtower service (and without requiring them to be online more than once every three months) **[Law22a]**,
7. arbitrarily small payments are routed securely, provided the parties routing the payment are self-interested,
8. there is never any incremental on-chain footprint required to resolve a payment, and
9. all payments are resolved within seconds.

The OPR protocol's properties (plus its simplicity) make it promising for small everyday payments.

10 Implementation Issues

10.1 Base Funds

Before a channel can be used for routing OPR payments, both parties must put base funds into the burn output. If the channel is dual-funded, this can be accomplished by putting both parties' base funds in the burn output in the initial channel state. If the channel is single-funded, the first use of the channel must be a payment to the party that did not fund the channel in order to allow that party to contribute base funds.

Also, as noted in Section 9.1, an optimization can be made in which a portion of the base funds are used to pay the fees for putting the channel state on-chain.

10.2 Larger Payments

While the OPR protocol is ideal for small payments, it can also support large payments.

Single Large Purchase

There is no limit on the size of a purchase that can be made as a single OPR payment. The main costs of using the OPR protocol are:

- risk of HTLC failure (due to the very short *htlc_expiry_delta_msec* value),
- risk of burning funds, and
- capital inefficiency (due to the need for base funds and matching funds).

In deciding which protocol to use, these costs have to be balanced against the following advantages of the OPR protocol:

- speed,
- watchtower-freedom for casual users,
- scalability (including ability to be used within a factory or timeout-tree),
- capital efficiency (due to the very fast resolution of payments), and
- lack of on-chain fees for payment resolution.

Alternatively, a large purchase can be made with multiple smaller payments that are performed sequentially and/or in parallel. For example, if nodes implement the current Lightning protocol with *cltv_expiry_delta* parameters of 34 blocks (approximately 20k seconds) and the OPR protocol with *htlc_expiry_delta_msec* parameters of 5k msec, 4k sequential OPR payment attempts can be made in the same worst-case time as a single payment attempt using the current Lightning protocol. Of course, the actual payment latency depends on the fraction of payment attempts that succeed and their average latency, but the law of large numbers means that the variance in the time to complete the OPR payments is much lower than the variance in the time to complete the payment using the current Lightning protocol.

Another factor of 5 could be achieved if 5 sets of 4k sequential OPR payment attempts are performed in parallel. Using this approach, a single $200k Lightning purchase using the current protocol could be replaced with 20k $10 OPR payments.

Recurring Payments

Another use of multiple small OPR payments is for recurring payments such as bills or mortgage payments. Many bills have a fixed size and are due once a month, so they could easily be replaced with small sequential OPR payments. Even bills that vary from month to month typically allow at least a week for payment, again allowing them to be paid with small sequential OPR payments.

Streaming Payments

A final category consists of services which could be paid for with a continuous stream of micro-payments. If the service is online, the low latency of a stream of OPR payments would allow the service to be turned on and off in a fine-grained manner that matches the payments received.

11 Griefer-Penalized Protocols For Other Applications

In the OPR protocol, both parties move funds to a burn output and they only receive those funds if they agree on how the funds should be divided. The technique of devoting funds to a burn output can be used for any protocol where both parties automatically calculate the same division of funds (at least in the vast majority of cases) and both parties always receive a significant portion of those funds (in order to motivate cooperation).

Using a burn output in this manner has many advantages over using pre-signed transactions to determine fund divisions, including:

- elimination of the need for calculating and pre-signing transactions,
- the ability to determine the correct division of funds in seconds,
- the ability to use computations that are not supported by Bitcoin script when dividing funds, and
- the lack of any on-chain footprint when dividing funds.

For example, burn outputs could be used as an alternative to Discreet Log Contracts **[Dryja]** for resolving smart contracts. This could be attractive as it eliminates the need for an oracle that creates a Schnorr signature, as well as the need to create a large number of pre-signed contracts **[Fournier22]**.

Another example is the creation of a fee-based protocol for reducing spam on Lightning. Such a protocol will be presented in a future paper.

12 Related Work

The OPR protocol's use of a burn output to obtain cooperation is similar to the use of security bonds. However, the OPR protocol differs by having both partners contribute to the burn output, and by apportioning burn output funds to the partners based on a condition that they calculate off-chain (as opposed to the result of an on-chain Bitcoin script). It is this difference that allows the OPR protocol to resolve payments so quickly and efficiently.

The idea of using a burn output to encourage cooperation between channel partners was suggested by Riard **[Riard22]** in the context of preventing channel jamming. However, Riard and Naumenko **[RN22]** left the creation of a protocol based on a burn output as future work.

13 Conclusions

The ability to support small, everyday payments is essential if Lightning is to become a widely-used means of payment. Although the OPR protocol has weaker security than the current Lightning protocol when making large payments, the comparison reverses with small payments.

The OPR protocol's ability to route small payments, coupled with its excellent scalability and usability, makes it an attractive candidate for supporting large numbers of casual Lightning users. The OPR protocol can also be used for large payments, either directly or by dividing the large payment into many smaller ones.

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

BOLT3 BOLT 03 - Transactions, https://github.com/lightning/bolts/blob/master/03-transactions.md

DRO18 Decker, Russell and Osuntokun, "eltoo: A Simple Layer2 Protocol for Bitcoin", https://blockstream.com/eltoo.pdf

Dryja Dryja, "Discreet Log Contracts", https://adiabat.github.io/dlc.pdf

Fournier22 Fournier, "CTV dramatically improves DLCs", https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-January/019808.html

Law21 Law, "Scaling Bitcoin With Inherited IDs", https://github.com/JohnLaw2/btc-iids

Law22a Law, "Watchtower-Free Lightning Channels For Casual Users" https://github.com/JohnLaw2/ln-watchtower-free

Law22b Law, "Lightning Channels With Tunable Penalties", https://github.com/JohnLaw2/ln-tunable-penalties

Law22c "Factory-Optimized Channel Protocols For Lightning", https://github.com/JohnLaw2/ln-factory-optimized

Law22d "Efficient Factories For Lightning Channels", https://github.com/JohnLaw2/ln-efficient-factories

Law23a Law, "Resizing Lightning Channels Off-Chain With Hierarchical Channels", https://github.com/JohnLaw2/ln-hierarchical-channels

Law23b Law, "Scaling Lightning With Simple Covenants", https://github.com/JohnLaw2/ln-scaling-covenants

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/

TF Transaction Fees, https://blockchain.com/explorer/charts/transaction-fees