# Watchtower-Free Lightning Channels For Casual Users

John Law

September 15, 2022

Version 1.0

**Abstract**

The Lightning Network offers users the potential for fast, low-fee and trust-free bitcoin payments. However, the network's high availability requirements cannot be met by casual users who could turn off or lose their phone for an extended period of time. As a result, casual users must outsource monitoring of the blockchain to a watchtower service, thus increasing their fees and eliminating freedom from trust. This paper presents a new channel protocol that allows casual users to send and receive Lightning payments without having to meet onerous availability requirements or use a watchtower service. This new Watchtower-Free (WF) protocol can also be used to simplify the reception of Lightning payments for casual users. No change to the underlying Bitcoin protocol is required.

## 1 Introduction

The Lightning Network **[AOP21][BOLT][PD16]** is the leading Layer 2 protocol **[GMRMG]** for scaling Bitcoin **[Nak09]**. In addition to improving scalability by allowing many Bitcoin transactions to remain off-chain in the normal case, Lightning improves the user experience by reducing payment latency and fees relative to using on-chain Bitcoin transactions.

Unfortunately, users of the Lightning Network are required to constantly monitor the blockchain and to respond to certain on-chain transactions by putting their own transaction(s) on-chain within a bounded time. The need to respond within a bounded time puts an availability requirement on the user that may be difficult to meet as the user's Lightning app must be open and running in the foreground **[Cor21a]** and the phone it is running on cannot be lost, broken, turned off or out-of-range for an extended period of time. As a result, casual users are encouraged to use a watchtower service **[GMRMG]** run by a third party that performs the required blockchain monitoring and on-chain responses. While a watchtower service can provide a casual user access to Lightning, the use of a watchtower service can increase the fees that the user pays and it eliminates the user's freedom from trust in making payments (for example, the watchtower service and the user's channel partner could collude to put an old channel state on-chain

and thus steal some of the user's funds). It would be far preferable for the casual user if a Layer 2 protocol accommodated their unavailability without requiring a watchtower.

The underlying Bitcoin protocol has always recognized that not all users can (or want) to run the full Bitcoin protocol, so it allows users to designate themselves as either full nodes (running the full protocol) or Simplified-Payment-Verification (SPV) nodes (running a stripped-down version of the protocol) **[Nak09]**. In a similar vein, the current paper allows Lightning users to designate themselves as either *Dedicated-Lightning-Users* (*DLUs*) or *Casual-Lightning-Users* (*CLUs*). CLUs can send and receive Lightning payments, but they cannot route Lightning payments for others. Thus, the capabilities of CLUs match the desires of most casual Lightning users.

This paper presents a new Lightning channel protocol that allows CLUs to send and receive Lightning payments without requiring a watchtower service and without having to meet onerous availability requirements. The new *Watchtower-Free* (*WF*) protocol can also allow a CLU to receive a payment without having to wait for additional blocks to be added to the blockchain, even in the worst case. No change to the underlying Bitcoin protocol is required.

The remainder of this paper is organized as follows. Section 2 reviews the current Lightning channel protocol and introduces notation used throughout the paper. Section 3 presents and analyzes the WF protocol, along with simplifications for CLUs receiving payments. Remaining sections discuss related work and present conclusions. A detailed timing analysis, including the settings for the protocol's timing parameters, is given in Appendix A.

# 2 The Current Lightning Channel Protocol

A Lightning channel **[PD16]** is operated by two parties who fund the channel and then update its state off-chain. The channel is created when a *Funding* transaction is put on-chain. The Funding transaction has a single output that requires signatures from both parties.

Each channel state is implemented by a pair of *Commitment* transactions, one of which is held by each party. Each Commitment transaction spends the output of the Funding transaction and has two outputs (one per party), each of which pays a portion of the channel's funds to the given party.

In addition, a Commitment transaction can have one or more *Hash Time-Locked Contract (HTLC)* **[BWHTLC]** outputs that control the distribution of funds for payments that are currently being processed by the channel. Each HTLC output implements an HTLC in which one party in the channel offers the other party a payment in exchange for the receiving party revealing a secret (namely the hash preimage of a given value X) before a specified expiry. In order to receive the funds from an HTLC output, the party receiving the payment can spend the HTLC output while revealing the secret to the other party. Alternatively, after the expiry specified in the HTLC, the party offering the HTLC can spend the HTLC output and return the funds in the HTLC to the offering party.

Signed transactions for old states are revoked by using revocation keys that allow the other party (that is, the party that did not put the old transaction on-chain) to receive all the funds in the channel. In normal operation, the parties update the channel state by creating and signing new Commitment, *HTLC-success* (which pay the party receiving the HTLC from that party's Commitment transaction) and *HTLC-timeout* (which pay the party offering the HTLC from that party's Commitment transaction) transactions and then exchanging revocation keys for the previous state. In this manner, the parties are able to update the channel state completely off-chain. If at any time the parties agree to close the channel, they can create a *Cooperative Close* transaction that spends the output of the Funding transaction and distributes the funds to the parties as defined by the final channel state.

Parties that do not share a channel can exchange bitcoin by making payments through a path of channels, where successive channels in the path include a common party. Each channel in the path is referred to as a *hop*. Multi-hop payments are made logically atomic by sharing a common secret (hash preimage), and this secret can be used as a *payment receipt* by the party initiating the payment.

When creating a channel, each party specifies a number of channel parameters, including a *to_self_delay* parameter and a *cltv_expiry_delta* parameter. The *to_self_delay* parameter is a safety parameter that specifies how long the other party must wait before they can receive funds from a transaction that they put on-chain (this delay is needed in order to provide a window during which an old transaction can be revoked). The *cltv_expiry_delta* parameter specifies how much later expiries for HTLCs in the given channel must be relative to those in the next hop.

An example of the current Lightning channel protocol is given in Figure 1, which shows a channel shared by Alice and Bob, with a single multi-hop payment initiated by Alice, where the next hop is in a channel shared by Bob and Carol.

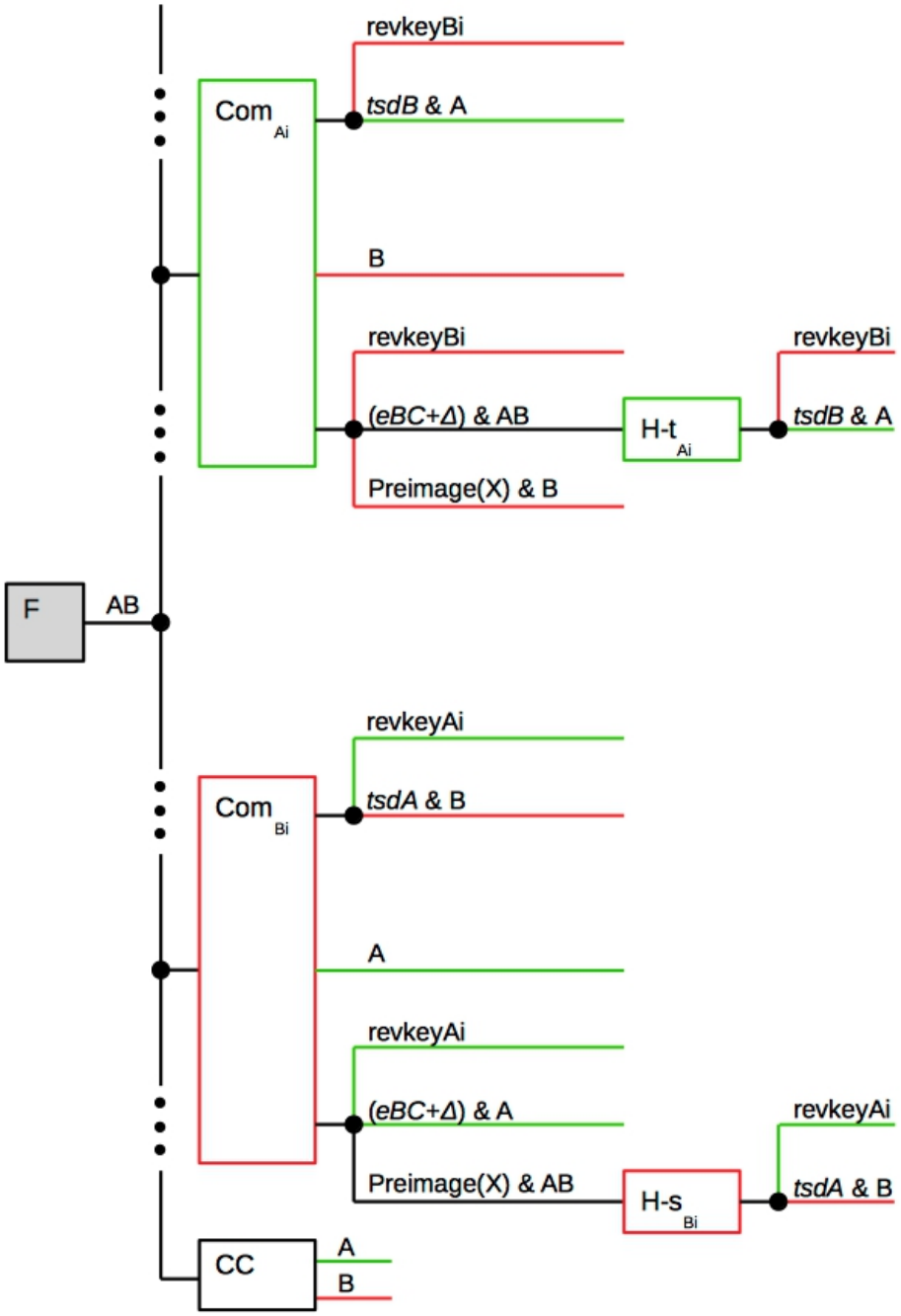


**Figure 1: The Current Lightning Channel Protocol.** The on-chain Funding (F) transaction has a single output that can be spent by a Cooperative Close (CC) or Commitment (Com) transaction. Alice has initiated a single multi-hop payment with the next hop being in a channel shared by Bob and Carol.

In Figure 1 (and throughout the paper):

- **A** denotes Alice's signature,
- **B** denotes Bob's signature,
- **AB** denotes both Alice's and Bob's signature,
- **revkeyAi** denotes a signature using a revocation key that allows Alice to revoke an old state *i* Commitment or HTLC-success transaction put on-chain by Bob[1],
- **revkeyBi** denotes a signature using a revocation key that allows Bob to revoke an old state *i* Commitment or HTLC-timeout transaction put on-chain by Alice,
- ***eBC*** denotes the expiry for this payment in the next hop shared by Bob and Carol,
- ***Δ*** denotes the *cltv_expiry_delta* parameter set by Bob,
- ***tsdA*** denotes the *to_self_delay* channel parameter set by Alice (controlling Bob's payments to himself),
- ***tsdB*** denotes the *to_self_delay* channel parameter set by Bob (controlling Alice's payments to herself), and
- **Preimage(X)** denotes the payment secret (and payment receipt) which is the preimage of X.

Throughout this paper, transactions are represented with boxes, outputs are represented by lines that leave a box from the right side, and inputs are represented by lines that enter a box from the left side. Shaded boxes represent transactions that are on-chain, while 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, and
- **CC** for a Cooperative Close 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 figure). 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.

1 For simplicity, the term "revkeyAi" (respectively, "revkeyBi") refers to a signature using any revocation key that allows Alice (respectively, Bob) to revoke an old state *i* Commitment, HTLC-success or HTLC-timeout transaction, even though different keys can be used by the same party to revoke different transactions for the same state.

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 (as each output can only be spent once). A party will be said to *submit* a transaction when they attempt to put it 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. Outputs labeled with "A" require Alice's signature, those labeled with "B" require Bob's signature, and those labeled with "AB" require both parties' signatures. 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.

For example, "*tsdB* & A" means that the given output case cannot be executed until a relative delay of *tsdB* (Bob's *to_self_delay* parameter) and that it requires Alice's signature. As another example, "(*eBC+Δ*) & AB" means that the given output case cannot be executed until *Δ* (Bob's *cltv_expiry_delta* parameter) after *eBC* (the expiry for this payment in the next hop)[2], and that it requires signatures from both Alice and Bob.

As can be seen in Figure 1, if either party puts an old Commitment transaction (and potentially an old HTLC-timeout or HTLC-success transaction that spends an HTLC output of that old Commitment transaction) on chain, the other party can use revocation keys to prevent the party that put the old transaction(s) on-chain from obtaining any of the funds in the channel.

Additional information about the Lightning channel protocol can be found in **[AOP21][BOLT][PD16]**.

# 3 The Watchtower-Free (WF) Protocol

## 3.1 Properties

The user-visible properties of the WF protocol can be expressed using two parameters:

- $\boldsymbol{I_S}$ which is a short time interval (e.g., 10 minutes) for communicating with peers, checking the blockchain, and submitting transactions, and
- $\boldsymbol{I_L}$ which is a long time interval (e.g., 1-3 months).

The casual user must be online for up to:

- $I_S$ every $I_L$ (e.g., 10 minutes every 1-3 months) to safeguard the funds in their Lightning channel.

2 While the label shows what conditions must be met for the given case, it does not show how they are enforced. In the HTLC output from Alice's state *i* Commitment transaction, the HTLC expiry (*eBC+Δ*) is implemented in the nLocktime field of the spending HTLC-timeout transaction. In contrast, in the HTLC output from Bob's state *i* Commitment transaction the HTLC expiry is implemented in both the output script in the Commitment transaction (using a CHECKLOCKTIMEVERIFY operator) and the nLocktime field of the spending transaction. The output script in Alice's Commitment transaction does not require the CHECKLOCKTIMEVERIFY operator, because it also requires Bob's signature (to ensure that the output of Alice's HTLC-timeout transaction has the required revocation key and relative delay), and Bob only signs HTLC-timeout transactions with the correct nLocktime value.

With the WF protocol, the latency for payments is unchanged from the current protocol, but the latency for getting a payment receipt from an uncooperative channel partner is increased. In addition, the casual user may have to pay their channel partner for the partner's cost of capital (which depends on $I_L$).

If the casual user and their channel partner follow the protocol, the channel can remain off-chain arbitrarily long.

## 3.2 First Attempt: Use the Current Lightning Channel Protocol

In order to motivate the new protocol, first consider what would happen if a casual user attempted to achieve the above properties with the current Lightning channel protocol. The casual user would set their *to_self_delay* and *cltv_expiry_delta* parameters to values approaching $I_L$ (because the casual user could be unavailable for nearly that long).

This would create three problems:

- **Problem 1**: The casual user's proposed channel partner would likely reject the creation of the channel due to the excessive *to_self_delay* value.
- **Problem 2**: If a channel were created with these parameters, Lightning payments would not be routed through the channel due to the excessive *cltv_expiry_delta* value.
- **Problem 3**: If a channel were created with these parameters and if the casual user sent a payment on that channel, their partner could have to go on-chain in order to pull the payment from the casual user. In particular, the casual user could be offline for nearly $I_L$ (e.g., 1-3 months) when their partner receives the receipt, thus forcing their partner to go on-chain to receive payment before the expiry of the associated HTLC.

## 3.3 The WF Protocol

### *Overview*

The WF protocol solves these problems by modifying the Lightning protocol as follows:

- **Problem 1** is solved by having the casual user pre-pay their channel partner for the cost of the partner's capital that is tied up in the channel due to the very large *to_self_delay* value. This pre-payment is included in the initial channel state and is updated at least once every $I_L$ to reflect the additional cost of capital due to the partner not yet going on-chain (assuming that is the case). If the casual user fails to update the state to reflect a new pre-payment, their partner can close the channel and receive the existing pre-payments.
- **Problem 2** is solved by allowing casual users to designate themselves as *Casual-Lightning-Users (CLUs)*, while the remaining users are *Dedicated-Lightning-Users (DLUs)*. CLUs can only partner with DLUs to open channels, such channels must be unannounced, and CLUs must not route (as opposed to send or receive) payments.

- **Problem 3** is solved by modifying both users' Commitment transactions in the channel that is sending the payment so the CLU can be offline for nearly $I_L$ without forcing the DLU to go on-chain. A simple approach would be to delay the expiry of the HTLC for each payment in the sending channel by $I_L$. That approach works, but has the downside of delaying (by $I_L$) the CLU's ability to force production of a payment receipt. A better approach is to add a relative delay before the CLU can time out the HTLC output of a Commitment transaction, thus enabling the DLU to safely stay off-chain even after the expiry of the HTLC. That is the approach taken here.

## *Protocol Specification*

The protocol will be specified in terms of a CLU Alice who shares a channel with DLU Bob. Bob sets his channel parameters as he would in the current Lightning protocol, while Alice sets her *to_self_delay* parameter (controlling Bob's payments to himself) to $I_L$ greater than it would be in the current Lightning protocol. Consider the case where Alice sends a Lightning payment on the channel she shares with Bob, with the next hop being on a channel shared by Bob and Carol. Let $eAB = eBC+\Delta$ denote the expiry of the HTLC in the channel shared by Alice and Bob.

Three changes are made relative to the current Lightning protocol:

- a relative delay of *tsdB* is enforced before Alice can spend the HTLC output for this payment in either Commitment transaction,
- after $eAB = eBC+\Delta$, only Alice's (rather than both parties') signature is required to spend the HTLC output in Alice's Commitment transaction and that output does not need to be spent using an HTLC-timeout transaction that can be revoked[3], and
- both parties update the channel state off-chain at least once every $I_L$ to reflect Bob's cost of capital, as described above.

The resulting protocol, with a single payment from Alice outstanding, is shown in Figure 2.

3 This change is possible because the relative delay added above guarantees that Bob can use his revocation key to prevent Alice from spending the HTLC output in an old Commitment transaction that she puts on-chain. As a result, Bob's signature is no longer required to ensure the use of such an HTLC-timeout transaction. Note that this change implies that the HTLC expiry *eAB* is implemented in both the output script in Alice's Commitment transaction (using a CHECKLOCKTIMEVERIFY operator) and the nLocktime field of the spending transaction.

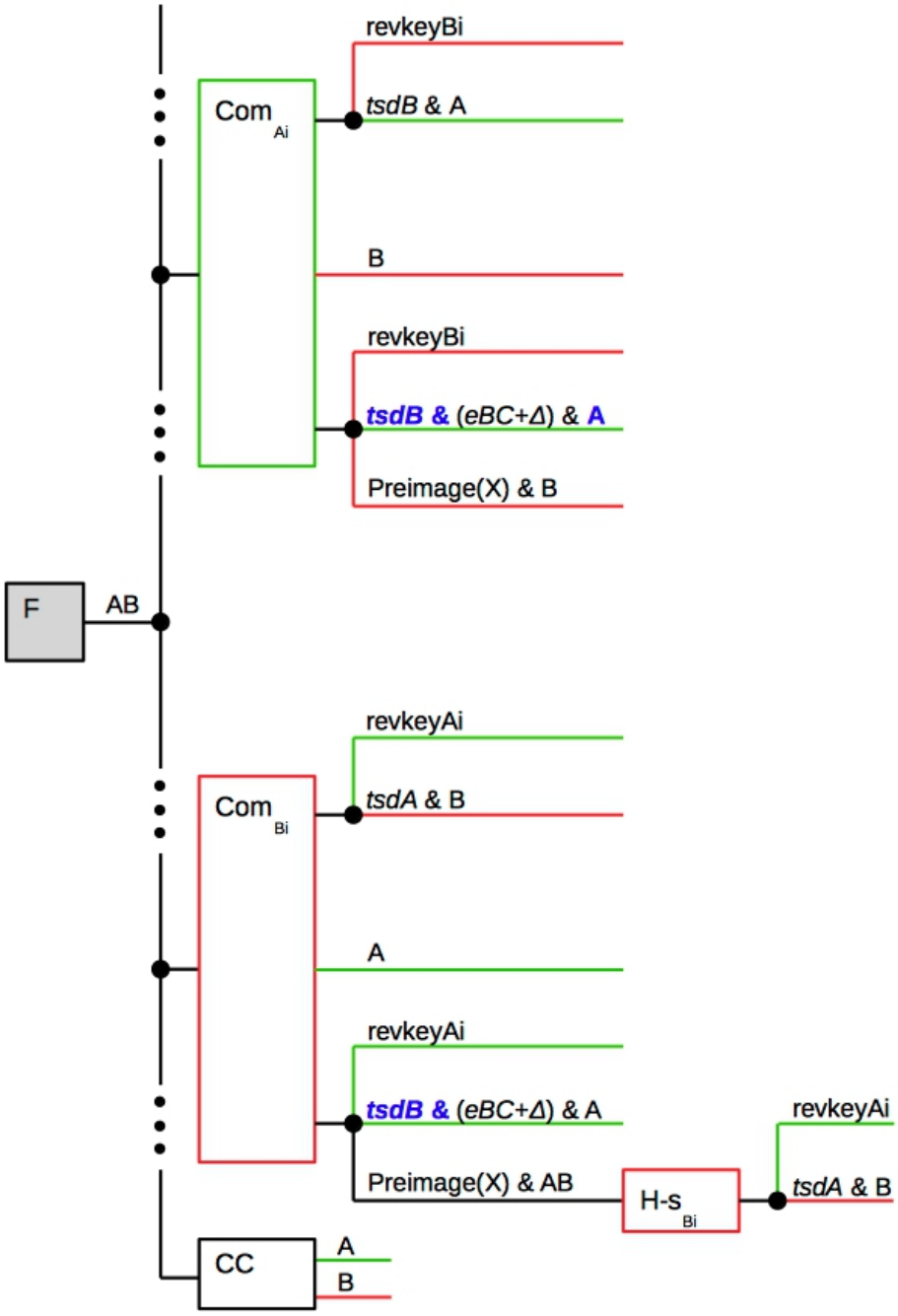


**Figure 2: The WF Channel Protocol.** Changes relative to the current Lightning channel protocol are highlighted in blue. CLU Alice must wait a relative delay equal to DLU Bob's *to_self_delay* (*tsdB*) before she can spend the HTLC output for the payment that she initiated, thus allowing Bob to remain off-chain even after the expiry of the HTLC.

When Bob receives Preimage(X) in the channel he shares with Carol, he sends Preimage(X) to Alice and attempts to update both parties' Commitment transactions to show payment of the HTLC. If he has spent $I_L$ time unsuccessfully trying to update those Commitment transactions, he can submit his Commitment and HTLC-success transactions. If at any point he sees Alice's Commitment transaction on-chain, he immediately stops trying to update the Commitment transactions off-chain and he submits his transaction that spends the HTLC output in Alice's Commitment transaction and reveals Preimage(X).

Alice implements the WF channel protocol as she would the current Lightning channel protocol, except:

- she can choose to be intentionally unavailable, provided she is available (or at least not intentionally unavailable) for at least $I_S$ every $I_L$ (to update her pre-payment for Bob's cost of capital and to revoke any old transactions put on-chain by Bob), and
- she does not submit her Commitment transaction until she has been available (or at least not intentionally unavailable) for at least a grace period of $G$ following the expiry of her offered HTLC (namely, *eAB*)[4].

### *Correctness*

The correctness of the WF protocol depends on showing that if Bob pays the HTLC (and thus receives Preimage(X)) in the next hop, then Bob will receive the HTLC payment in the current hop. There are four cases:

- **Case 1:** Bob sees Alice's Commitment transaction on-chain when he receives Preimage(X). In this case, Bob's transaction that reveals Preimage(X) and spends the HTLC output in her Commitment transaction wins the race against Alice's conflicting transaction due to the definition of Bob's *cltv_expiry_delta* parameter (Δ), which is large enough to guarantee this outcome[5].
- **Case 2:** Not Case 1, and Bob's Commitment transaction is in the final version of the blockchain. In this case, Bob's HTLC-success transaction wins the race against Alice's conflicting transaction due to the definition of Bob's *to_self_delay* parameter (*tsdB*).
- **Case 3:** Not Case 1, and Alice's Commitment transaction is in the final version of the blockchain. In this case, Bob's transaction that reveals Preimage(X) and spends the HTLC output in Alice's Commitment transaction wins the race against Alice's conflicting transaction due to the definition of Bob's *to_self_delay* parameter (*tsdB*).
- **Case 4:** Not Case 1, 2 or 3. In this case, it follows that Bob was able to update the channel state off-chain.

4 This grace period of $G$ is the same as the one defined for the current Lightning protocol, and $G \leq I_S$.
5 The required values of Bob's *cltv_expiry_delta* and *to_self_delay* parameters will be established in Appendix A.

Therefore Bob receives the HTLC payment from Alice if he makes the HTLC payment to Carol.

We will now show that neither Alice nor Bob will go on-chain if both parties follow the protocol. First, consider the case where Bob makes the HTLC payment to Carol. In this case, during the $I_L$ time after Bob receives Preimage(X), Alice will have a period of at least $I_S \geq G$ during which she is not intentionally unavailable, and $G$ is long enough for Alice and Bob to update the channel state off-chain. Therefore, Bob will not give up and submit his Commitment transaction. Furthermore, Alice will not submit her Commitment transaction until she has been available (or at least not intentionally unavailable) for a grace period of $G$ after the expiry of her offered HTLC (and thus after Bob has received Preimage(X)), and $G$ is long enough to update the channel state off-chain. Therefore, Alice will not give up and submit her Commitment transaction.

The case where Bob does not make the HTLC payment to Carol is similar, except the state in the channel shared by Alice and Bob is updated off-chain to indicate that the HTLC timed out.

### *Fees*

If the fees in a signed off-chain transaction are insufficient when that transaction is submitted, the transaction can be submitted as part of a package where the transaction's child provides the required additional fees **[Zha22]**. There are situations in the WF protocol where a party submits parent and child transactions together (such as when Bob submits his Commitment and HTLC-success transactions), so the child-with-unconfirmed-ancestors form of the package relay proposal would be most convenient, but the child-with-unconfirmed-parents form could be used if the fee-paying child is made a direct child of all other transactions in the package.

## 3.4 One-Shot Receives

### *Motivation*

While eliminating watchtowers is helpful for casual users, the protocol for receiving Lightning payments is still awkward for such users. With the current Lightning protocol, when a user receives a payment and their channel partner is unresponsive, the user must submit their Commitment and HTLC-success transactions. However, if their partner's conflicting Commitment transaction wins the race and is included in the blockchain, the user then has to submit a different transaction that reveals the HTLC's secret and spends the HTLC output in their partner's Commitment transaction.

An example of the current Lightning channel protocol is given in Figure 3, which shows a channel shared by Alice and Bob, with a single multi-hop payment to Alice that is routed through Bob. In this figure (and throughout the paper):

- ***mfceA*** denotes the *min_final_cltv_expiry* parameter set by Alice.

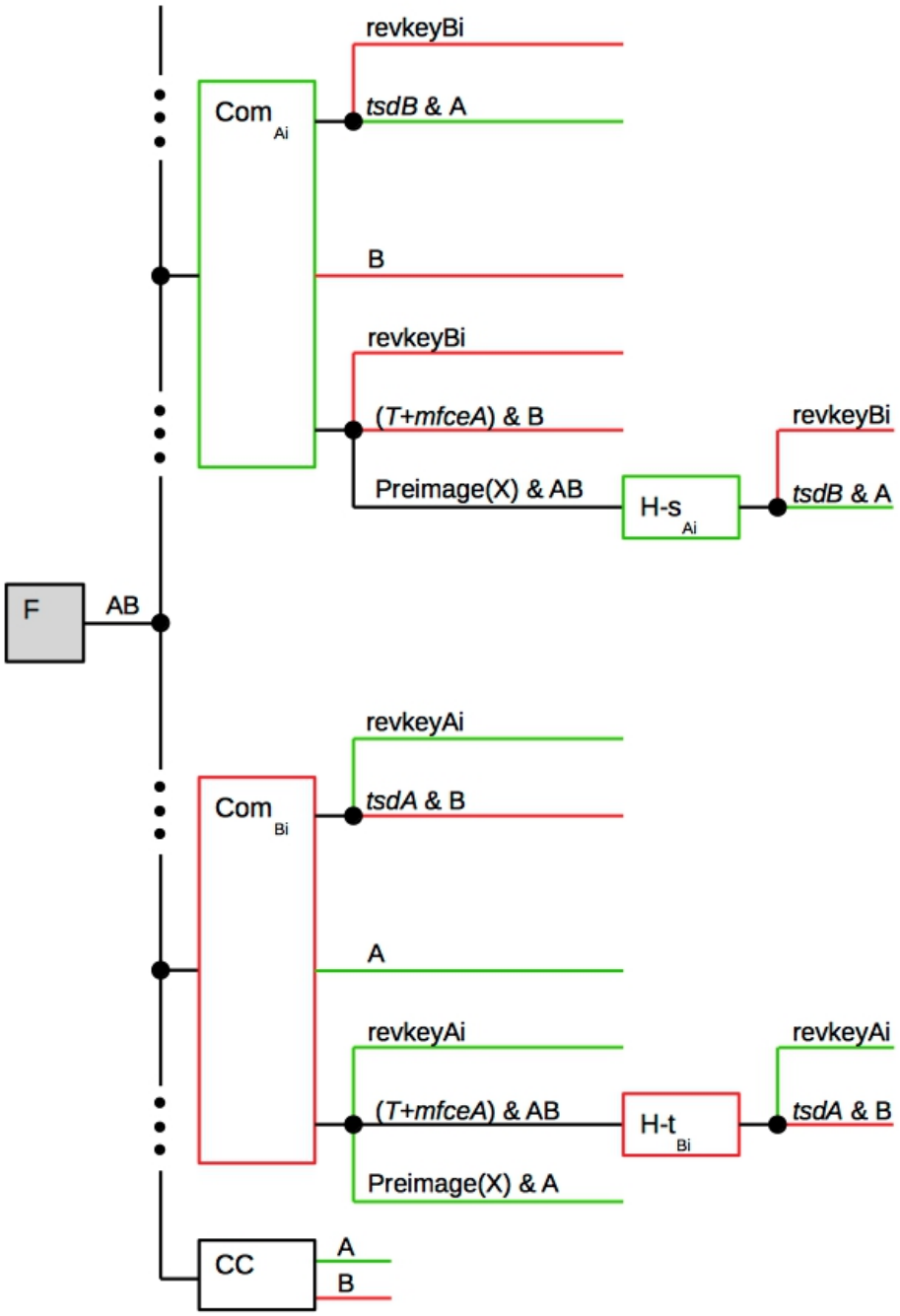


**Figure 3: The last hop of the current Lightning protocol.** Alice can attempt to make the HTLC succeed by submitting her Commitment and HTLC-success transactions. However, if Bob's conflicting Commitment transaction wins the race, Alice must then submit another transaction that spends the HTLC output from Bob's Commitment transaction and provides Preimage(X).

The requirement to wait and check the blockchain for the winning Commitment transaction (which might not be determined until multiple blocks have been added to the blockchain) is undesirable. It would be far preferable if the casual user could always receive a payment by performing a sequence of off-chain message exchanges and at most one submission of transactions. A protocol that has this property will be said to support *one-shot receives*. More generally, a procedure that requires performing actions at $X$ different blockheights will be called an *X-shot procedure*.

### *Implementation*

The modification of the WF protocol to support one-shot receives will be specified in terms of a CLU Alice who shares a channel with DLU Bob. The following change to the WF protocol is made whenever a new Commitment transaction for Bob is signed by Alice:

- if Bob has one or more outstanding HTLCs offered to Alice, the nLocktime field of Bob's Commitment transaction is set to the expiry of the earliest such HTLC,
- otherwise, the nLocktime field of Bob's Commitment transaction is set to $I_L$ in the future (relative to when Bob's Commitment transaction is signed by Alice).

Unfortunately, in the current Lightning protocol the nLocktime field in the Commitment transaction is set to a value in the past that provides 24 bits of the channel's state number in order to allow efficient revocation of old on-chain Commitments (with the remaining 24 bits being provided by the nSequence field of the Commitment transaction's sole input). Because support for one-shot receives requires the nLocktime field to enforce an absolute timelock, those 24 bits of state number can no longer be encoded in the nLocktime field. There are two solutions to this problem:

- add a second input to Bob's Commitment transaction that spends a UTXO owned by Bob (the value of which is arbitrary and is refunded to Bob in the Commitment transaction) and use the nSequence field of that input to encode 24 bits of state number, or
- support only 24-bit state numbers, as 16 million channel states are likely sufficient for most casual users.

These two solutions are shown in Figures 4 and 5, respectively, which show the channel state with a single outstanding multi-hop payment to Alice that is routed through Bob. Changes relative to the current Lightning protocol are shown in blue.

In these figures:

- $\mathbf{In_B}$ is an Individual transaction with a UTXO owned by Bob, and
- ***T*** denotes the time when the earliest currently-outstanding HTLC offered by Bob to Alice was created.

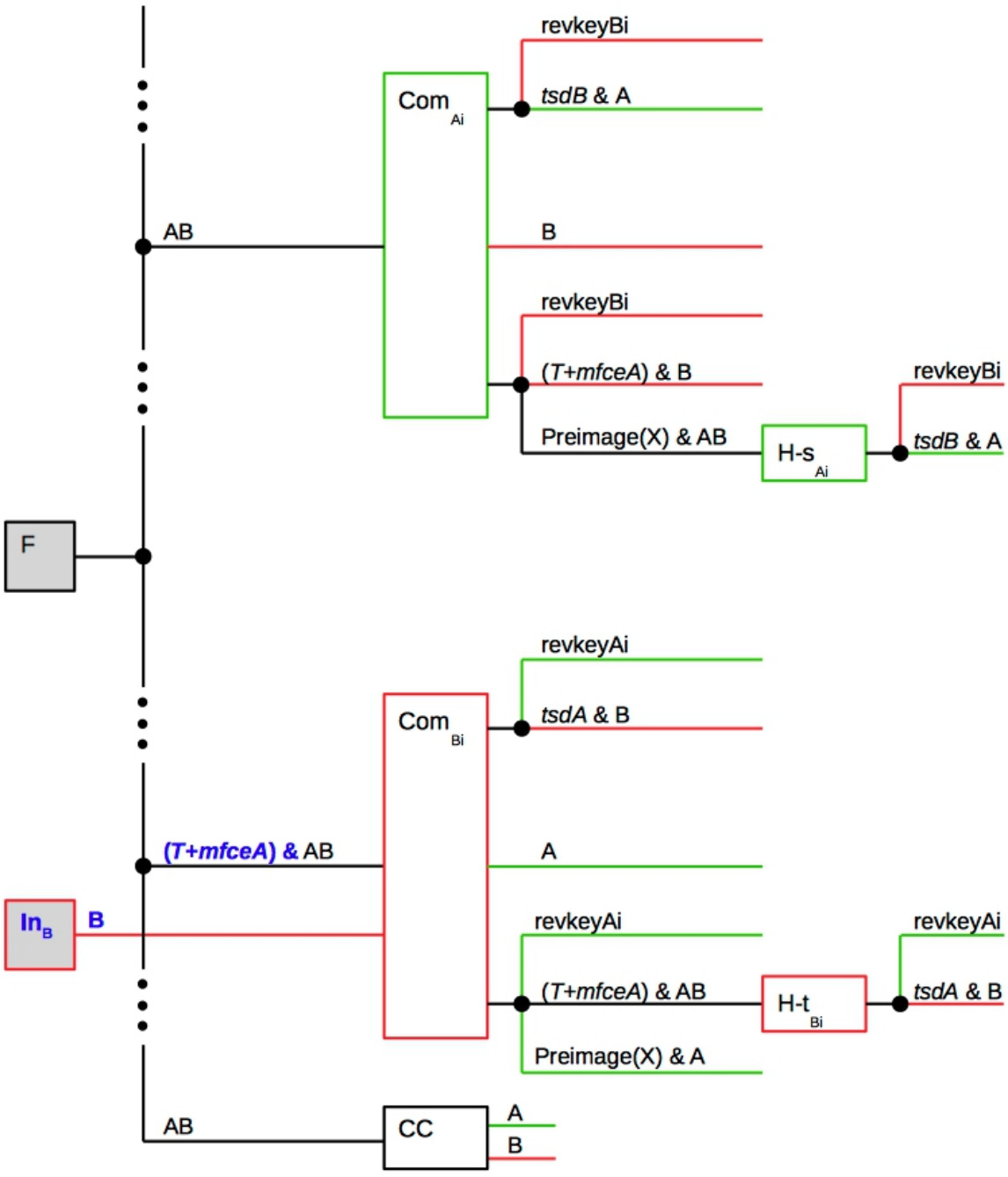


**Figure 4: One-shot receives with a two-input Commitment transaction.** The nLocktime field in DLU Bob's Commitment transaction is set to guarantee that CLU Alice can put her current Commitment transaction on-chain first. The 24-bit state hint that is normally included in that nLocktime field is moved to the second input's nSequence field. The first input in Bob's Commitment transaction requires both parties' signatures (using the SIGHASH_ALL flag), so those signatures guarantee that the second input and its nSequence state hint are as required.

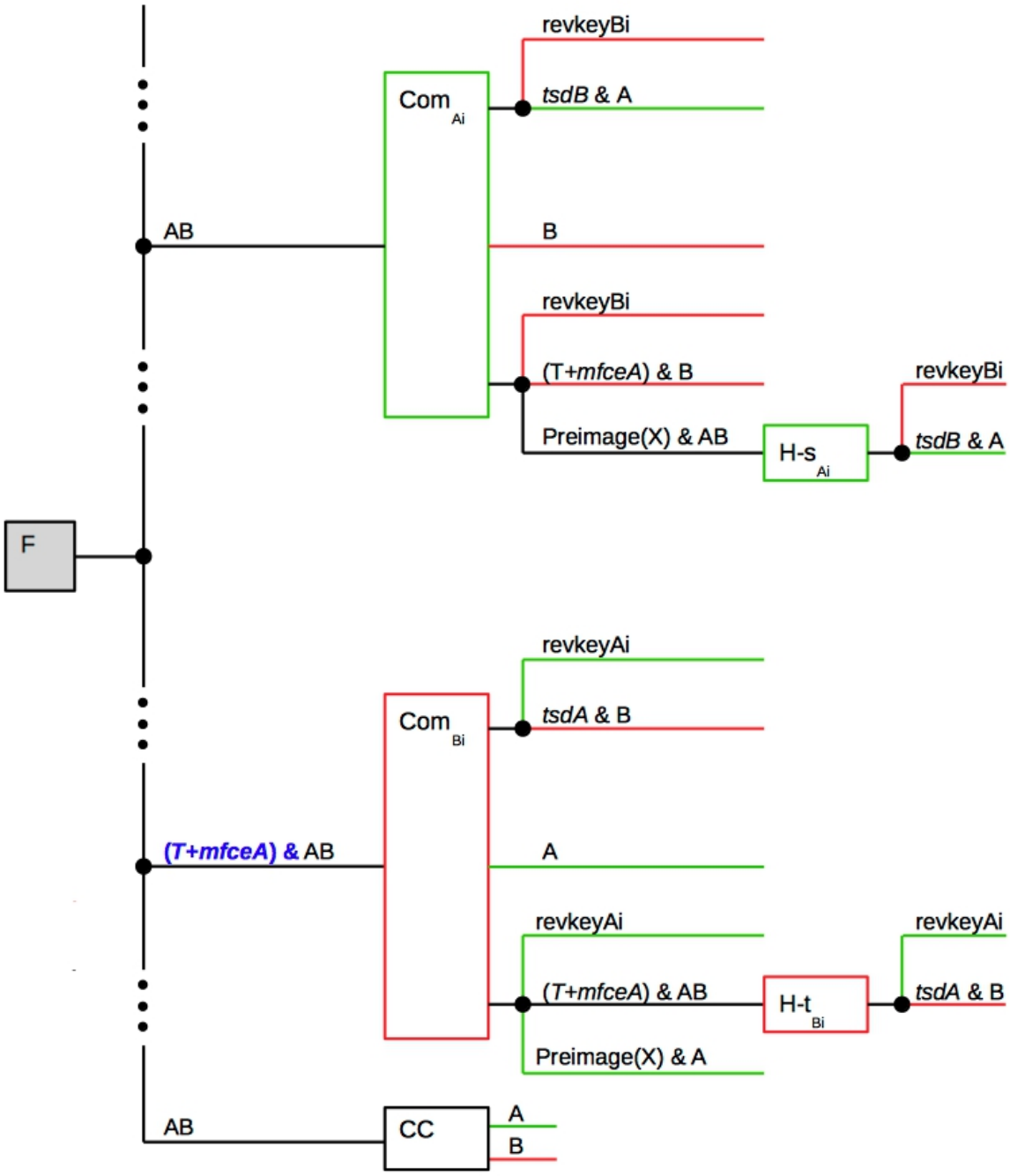


**Figure 5: One-shot receives with a one-input Commitment transaction.** The nLocktime field in DLU Bob's Commitment transaction is set to guarantee that CLU Alice can put her current Commitment transaction on-chain first. The 24-bit state hint that is normally included in that nLocktime field is removed, so only $2^{24}$ (approximately 16 million) channel states are supported.

In addition, the following constraints are added in order to guarantee one-shot receives:

1. Whenever a new HTLC is offered to Alice, its expiry is set to exactly *mfceA* in the future. This constraint guarantees that new HTLCs have expiries that are monotonically nondecreasing.
2. Whenever Alice gives Bob a secret for an HTLC, that HTLC has the earliest expiry of all the HTLCs in Alice's current Commitment transaction.
3. Whenever a new channel state $i$+1 is created, Alice's partial signature for Bob's Commitment transaction for state $i$+1 is given to Bob, and the revocation key for Bob's Commitment transaction for state $i$ is given to Alice, before Bob's partial signature for Alice's Commitment transaction for state $i$+1 is given to Alice.

### *Correctness*

Whenever Alice has given Bob a secret for an HTLC and she has been unable to update both parties' Commitment transactions off-chain to reflect the success of that HTLC, she submits a package consisting of her current Commitment transaction and her HTLC-success transaction for that HTLC early enough to guarantee that they will be on-chain prior to the expiry of the HTLC. Let $h$ denote the HTLC for which Alice is going on-chain and let $i$ denote the state of her Commitment transaction that she submits to the blockchain. Note from Constraint 2 that $h$ has the earliest expiry of any HTLC in Alice's Commitment transaction, and from Constraint 3 that Bob can only have an unrevoked Commitment transaction for state $i$ and/or state $i$+1. If Bob has an unrevoked Commitment transaction for state $i$, it follows that its nLocktime field is set to the expiry of $h$. If Bob has an unrevoked Commitment transaction for state $i$+1, it follows from Constraint 1 that its nLocktime field is set to the expiry of $h$ or later. Thus, it follows that Alice's Commitment and HTLC-success transactions will be on-chain before Bob can submit an unrevoked Commitment transaction. As a result, Alice will win the race in getting her current Commitment transaction on-chain[6], thus providing one-shot receives.

Finally, it is important to verify that the delay of Bob's Commitment transaction (caused by the setting of its nLocktime field) does not create any problems for Bob. First, for HTLCs offered to Alice (that is, payments received by Alice), the current Lightning protocol requires that Bob wait until after the expiry of this offered HTLC before he submits his Commitment and HTLC-timeout transactions. Therefore, the nLocktime field has no impact on Bob's actions regarding HTLCs offered to Alice. Second, for HTLCs offered by Alice (that is, payments sent by Alice), the WF protocol does not force Bob to submit his Commitment and associated HTLC-success transactions before any specific time in order to guarantee the success of any HTLCs[7]. As a result, Bob's ability to force payment for HTLCs offered by Alice is unaffected by the nLocktime field in this Commitment transactions.

---

6 Note, however, that Bob could put an old (revoked) Commitment transaction on-chain at any time. That case is handled by having Alice revoke his old Commitment transaction when she checks the blockchain at least once every $I_L$ prior to updating the channel state to reflect a new pre-payment.

7 The Lightning protocol does not have this property, which is why the changes described here cannot be made to the Lightning protocol in order to support one-shot receives.

## 3.5 Getting a Payment Receipt

Consider again the case where Alice has offered an HTLC to Bob. At any time after the expiry of the HTLC, if Alice needs to get a payment receipt and Bob is uncooperative, Alice can put her Commitment transaction on-chain and then attempt to spend the HTLC output of her Commitment transaction *tsdB* later. As was shown above, she is guaranteed to win the race in putting her Commitment transaction on-chain due to the nLocktime field in Bob's Commitment transaction (unless Bob's Commitment transaction has been revoked, in which case Alice will receive all of the channel's funds). Therefore, she will either get her receipt before she is able to spend the HTLC output or she will not have to make her payment (because she succeeded in spending the HTLC output). This procedure for getting a payment receipt is not one-shot and may be somewhat awkward for casual users. Fortunately, it is only required when there is both a payment dispute (or other need to get a receipt quickly) and an uncooperative channel partner.

## 3.6 Asynchronous Payments

The WF protocol provides great flexibility to CLUs, as they can perform one-shot sends and receives when desired, and the required channel maintenance can be performed at any time of their choosing within a large time window. However, there is still a constraint on when CLUs need to be online: if a CLU sends a payment to another CLU, they must both be online simultaneously. This constraint is natural in some settings, such as face-to-face payments, but it could be problematic in other settings, such as sending gifts and remittances. As a result, it would be beneficial if CLUs could send and receive bitcoin without having to synchronize their availability. Fortunately, the WF protocol can be modified quite easily to provide asynchronous payments.

### *Sender Has An Invoice With A Unique Secret*

First, consider the case where the sender has an invoice from the destination that requires a unique secret selected by the destination (for example, this invoice could have been sent in an email to the sender). Assume CLU Alice has a WF protocol channel with DLU Bob, and Alice wants to send bitcoin asynchronously to CLU Dave, who has a WF protocol channel with DLU Carol. Alice and Bob create an HTLC for the payment as defined by the WF protocol, with a relative delay of *tsdB*[8], but without an absolute delay of *eAB*, for Alice's ability to time out the HTLC.

Then, rather than having Alice send an onion message to Dave which establishes the payment's HTLCs, Alice sends a Trampoline payment **[Tra]** through Bob and asks him to hold it until he receives an onion message with nonce A **[Cor21a]**. Alice then sends an onion message to Carol which includes a reply onion to Bob with nonce A and asks Carol to send the reply onion when Dave is online. When Bob receives the reply onion with nonce A, he determines if he can make the payment and receive its payment receipt within *tsdB*. If he can do so, Bob sends the Trampoline payment that establishes the

8 Rather than using Bob's *to_self_delay* parameter, a separate delay parameter could be agreed upon for performing such asynchronous sends.

payment's HTLCs and performs the payment[9]. Even if Alice puts her current Commitment transaction on-chain, the *tsdB* relative delay before she can obtain a refund of the HTLC guarantees that Bob will be able to receive payment from Alice or he will not have to make payment in the next hop.

On the other hand, if Bob cannot guarantee that the payment will be made or timed out within *tsdB* (or if at any time Alice wants to cancel the payment), Alice and Bob update their channel state to refund the HTLC. Finally, even if Bob is not cooperative in canceling an asynchronous payment, Alice can force the payment to either occur or time out by putting her Commitment transaction on-chain.

Note that the resulting protocol is trust-free, as Alice only makes the payment if she receives a payment receipt, and Dave always receives the payment once he reveals his payment receipt. It is also efficient, as it does not require that capital be allocated by DLUs that implement the payment until Dave is online. It does require that Alice's capital be allocated in an HTLC until the payment is made or canceled, but that is capital that she wants to send and that she can recover at any time (with a delay of *tsdB*) if she changes her mind. Finally, note that the asynchronous payment offer can stay in effect for an arbitrary length of time without requiring any party to go on-chain, as it is based on a relative delay (in Alice's and Bob's Commitment transactions) rather than an absolute one.

### *Sender Does Not Have An Invoice With A Unique Secret*

There are some situations, such as tipping, in which it is difficult for the sender to asynchronously obtain an invoice from the receiver which requires a unique secret chosen by the receiver. If PTLCs are supported, the protocol created by Corallo can be used to add a secret created by the sender to a secret created by the receiver which is published using lnurl **[Cor21a]**, thus creating a unique secret for the payment. Once a unique secret has been created, it can be used with the protocol presented above to make the asynchronous payment.

# 4 Related Work

The WF protocol presented here is based extensively on previously-published work, namely the Poon-Dryja Lightning channel protocol **[PD16]** and the BOLT specifications **[BOLT]**. The technique given in Section 3.6 for asynchronously generating a unique secret is from Corallo **[Cor21a]**. Corallo also proposed using an extremely long CLTV in the first hop in order to facilitate asynchronous payments, which is somewhat similar to the WF protocol's allowance for the sender to be offline for a long period in order to support watchtower-freedom. The WF protocol differs by using a relative delay to allow the parties to remain offline past the expiry of the HTLC, while still allowing the sender to force their partner to produce a receipt (or not make payment) any time after the HTLC's expiry.

9 This Trampoline payment from Bob will occur at a time that is unknown when Alice sends the Trampoline payment to Bob. As a result, it is necessary to allow Bob to set the expiries of the payment's HTLCs, as suggested by Corallo **[Cor21b]**.

The idea of eliminating watchtowers for a casual user by delaying their partner's ability to put transactions on-chain was described by Law **[Law21]**, but the interaction of that delay with HTLCs was not analyzed and that paper assumed modifications to the underlying Bitcoin protocol.

# 5 Conclusions

This paper presents the idea of dividing users into Casual-Lightning-Users (CLUs) that only send and receive payments, and Dedicated-Lightning-Users (DLUs) that can also route payments. It gives a new protocol that allows CLUs to send and receive Lightning payments in a trust-free manner without using a watchtower service. It also allows CLUs to receive payments in a one-shot manner. Finally, it supports asynchronous payments between CLUs. No changes to the Bitcoin protocol are required.

The new protocol does have some disadvantages, such as increasing the cost of capital for DLUs that partner with CLUs and increasing the latency for CLUs to get payment receipts from uncooperative partners. Hopefully, the elimination of watchtowers for casual users, and their ability to do one-shot receives, more than make up for these drawbacks.

# References


**AOP21** Andreas Antonopoulos, Olaoluwa Osuntokun and Rene Pickhardt. Mastering the Lightning Network, 1st. ed. 2021.

**BOLT** BOLT (Basis Of Lightning Technology) specifications. See https://github.com/lightningnetwork/lightning-rfc.

**BWHTLC** Bitcoin Wiki: Hash Timelocked Contracts. See https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts.

**Cor21a** Matt Corallo. A Mobile Lightning User Goes To Pay A Mobile Lightning User.... See https://lists.linuxfoundation.org/pipermail/lightning-dev/2021-October/003307.html..

**Cor21b** Matt Corallo. A Mobile Lightning User Goes To Pay A Mobile Lightning User.... See https://lists.linuxfoundation.org/pipermail/lightning-dev/2021-October/003330.html.

**GMRMG** Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry and Arthur Gervais. SoK: Layer-Two Blockchain Protocols. See https://eprint.iacr.org/2019/360.pdf.

**Law21** John Law. Scaling Bitcoin With Inherited IDs (Section 3.6). See https://github.com/JohnLaw2/btc-iids.

**Nak09** Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2009. See http://bitcoin.org/bitcoin.pdf.

**PD16** Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments (Draft Version 0.5.9.2). January 14, 2016. See https://lightning.network/lightning-network-paper.pdf.

**Tow21** Anthony Towns. Lightning over taproot with PTLCs. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2021-October/003293.html

**Tra** Trampoline Payments. See https://bitcoinops.org/en/topics/trampoline-payments.

**Zha22** Gloria Zhao. Package Relay Proposal. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-May/020493.html.

# Appendix A: Timing Analysis

This appendix provides a detailed analysis of the timing parameters and deadlines used by the current Lightning channel protocol and the WF protocol presented above.

## A.1 Timing Model

This subsection presents the model that will be used for analyzing the timing of different protocols. It is based on the timing model used in the BOLT #2 specification **[BOLT]**, but it extends and adds detail to that model. In order to simplify the analysis, actions are modeled as taking a fixed worst-case time (rather than having a probability distribution).

The timing model used here makes the following assumptions and uses the parameters *R*, *S*, *G*, *B*, *U* and *L* as defined below:

1. Each user has a single version of the blockchain.
2. At all times, all users have versions of the blockchain with the same number of blocks in them.
3. At all times, all versions of the blockchain agree in all but the *R* most recent blocks.
4. Once a transaction is more than *R* blocks deep, it will appear in the same position in all current and future versions of the blockchain.
5. A user can attempt to add to the blockchain by submitting a package, consisting of an ordered list of one or more transactions, where:
   a) each transaction input spends a preceding transaction output or an unspent transaction output (UTXO) in that user's version of the blockchain, and
   b) each transaction's nLocktime and nSequence fields meet the requirements for inclusion in the next block of that user's version of the blockchain.
6. If a user submits a package according to a protocol's rules[10], then (*R*+*S*) blocks later, for every transaction in the package, either[11]:
   a) that transaction is more than *R* blocks deep in all versions of the blockchain, or

10 It is assumed that the protocol requires the use of competitive fees when submitting packages.
11 The *S* parameter includes any delays caused by unintentional unavailability of the user or their Bitcoin peers.

b) a transaction that conflicts with that transaction, or with an ancestor of that transaction, is more than *R* blocks deep in all versions of the blockchain.

7. In *G* time, a user is able to perform calculations, send and receive multiple Lightning protocol messages to and from peers, and determine the current height of the blockchain[12].
8. In *B* time, a user that has been intentionally unavailable is able to obtain a current version of the blockchain (including determining whether or not certain UTXOs and transactions exist in the blockchain)[13].
9. A user is unintentionally unavailable for a total of at most *U* time during their lifetime (and this unavailability may consist of a single outage of length *U* or it may consist of multiple shorter outages).

Because the sum *R*+*S* often arises in calculations, the latency parameter *L* will be defined as *L* = *R*+*S*.

Times will be expressed in terms of blocks, where the current time is the current blockheight plus 1, and the top block in the blockchain is said to be 1 block deep. A transaction will be said to be *fixed* if it is at least *R*+1 blocks deep (and thus will appear in that position in all versions of the blockchain), so a transaction that is fixed at time *T* is in block *T* - (*R*+1) or earlier.

Note that the parameters *S*, *G* and *B* include any delays caused by unintentional unavailability of the user or their peers, while the parameter *U* gives the longest period of unintentional unavailability that a user experiences. As a result, one could argue that *U* should be less than *S*, *G* and *B*, as the longest period of unintentional unavailability may occur when the user is attempting to submit a Bitcoin package (S), exchange Lightning messages with peers (G), or obtain a current version of the blockchain (B). However, in practice it makes sense to set *S*, *G* and *B* to values that are smaller than *U*, because:

- The parameter *U* includes the unintentional unavailability that a user experiences over a much longer time period than is covered by the parameters *S*, *G* and *B*.
- Although these parameters are modeled as being worst-case bounds, they are really probabilistic bounds that hold with high probability. Because a user may only spend a small fraction of time actively submitting Bitcoin packages, exchanging Lightning messages with peers, or obtaining a current version of the blockchain, it is likely that the user's worst period of unintentional unavailability will not affect the parameters *S*, *G* and *B*.

12 For example, a user can irrevocably add an HTLC output to both parties' Commitment transactions in *G* time by 1) sending an *update_add_htlc* and a *commitment_signed* message to their partner, 2) receiving a *revoke_and_ack* and a *commitment_signed* message in response, and 3) sending a *revoke_and_ack* message in response. As another example, in *G* time a user can calculate and establish a multi-hop route to initiate a new Lightning payment. The *G* parameter includes any delays caused by unintentional unavailability of the user or their Lightning peers.

13 The *B* parameter includes any delays caused by unintentional unavailability of the user or their Bitcoin peers. The current blockchain, UTXO and transaction information can be obtained using the SPV protocol if desired. Obtaining such a current version of the blockchain is only needed if the user was intentionally unavailable and thus not monitoring the blockchain.

- The parameters *S*, *G* and *B* determine protocol parameters that affect the safety of HTLCs, while the parameter *U* determines protocol parameters that affect the safety of all the funds in a Lightning channel. As a result, it makes sense to set the parameter *U* more conservatively[14].
- The parameters *S*, *G* and *B* determine protocol parameters that affect the safety of HTLCs that expire at times that are independent of the user's availability, while the parameter *U* determines channel safety that could be challenged when the user's channel partner detects that the user is unavailable (for example, by detecting the lack of a *pong* message in response to a *ping* message). As a result, it makes sense to set the parameter *U* more conservatively.

Example values of these parameters from the BOLT #2 specification and Lightning implementations are:

- R: 2 blocks,
- S: 12 blocks,
- G: 1 or 2 blocks, and
- U: 6*24*14 - (2*2+12+1) = 1999 blocks.

A plausible value for the B parameter is 1 or 2 blocks.

Because the G and B parameters refer to the time taken to perform certain off-chain activities, they can also be expressed in terms of wall time rather than numbers of blocks. Plausible wall-time values for G and B are 2 minutes and 8 minutes, respectively. These times are shorter than the expected time elapsed for 1 or 2 blocks to be added to the blockchain, but these wall-time values make sense because these parameters are actually bounds that hold with high probability, so expressing them as block counts has to include the rare possibility of blocks that are mined extremely quickly. For example, if the time to add a block to the blockchain is exponentially distributed with a mean of 10 minutes, the probability that a block will be added within 1 minute is $1 - e^{-0.1} \approx 0.095$.

## A.2 Timing Parameters in the Current Lightning Channel

The timing model presented above can be used to determine the timing parameters for the current Lightning channel protocol as follows:

### *timeout_deadline*

The *timeout_deadline* parameter gives the number of blocks after the HTLC expiry when the party offering the HTLC goes on-chain unless both parties' Commitment transactions are updated to reflect

14 The observation that the unintentional unavailability tolerated in order to secure HTLCs should be less than that tolerated to secure the safety of the channel funds was made earlier by Towns **[Tow21]**.

that the HTLC has timed out. The *timeout_deadline* will be set to *G* in order to allow the HTLC to be failed off-chain after its expiry, which matches its value in the BOLT #2 specification[15].

### *cltv_expiry_delta*

The *cltv_expiry_delta* parameter gives the number of blocks between the HTLC expiry in the next hop and the HTLC expiry in the current hop. The *cltv_expiry_delta* must be large enough to guarantee that if the HTLC is fulfilled in the next hop (with Bob and Carol) it will also be fulfilled in the current hop (with Alice and Bob). Starting with the HTLC expiry in the next hop, the worst case delay to guaranteed fulfillment of the HTLC in the current hop is as follows:

1. *G = timeout_deadline* before Bob gives up on the off-chain protocol and submits a package consisting of his Commitment and HTLC-timeout transactions in an attempt to time out the HTLC in the next hop.
2. *L* blocks before Carol's conflicting Commitment transaction wins the race with Bob's package and is fixed, at which point it is seen by Bob who submits his transaction that times out the HTLC.
3. *L* blocks before Carol's conflicting HTLC-success transaction wins the race with Bob's transaction and is fixed, at which point it is seen by Bob.
4. *G* time before Bob gives up on the off-chain protocol and submits a package consisting of his Commitment and HTLC-success transactions in an attempt to fulfill the HTLC in the current hop.
5. *L* blocks before Alice's conflicting Commitment transaction wins the race with Bob's package and is fixed, at which point it is seen by Bob, who then submits his secret-revealing transaction spending the HTLC output in Alice's Commitment transaction.
6. *L* blocks before Bob's secret-revealing transaction is fixed, at which point it is guaranteed to always remain in all versions of the blockchain.

The above analysis yields the following inequality:

$$cltv\_expiry\_delta \geq 2G + 4L = 2G + 4R + 4S.$$

15 Note that *timeout_deadline* could be set to *-L* (that is, the offering party submits their Commitment transaction *L* before the expiry) and *cltv_expiry_delta* could be reduced by *G + L* if the protocol for the fulfilling user were changed to require them to fail the HTLC (*G+fulfillment_deadline*-1) blocks before the HTLC expiry in the current channel (which is 1 block after the last time they could receive the secret for an HTLC that they paid in the next hop). In fact, the protocol for the fulfilling user *should* be changed to make them fail the HTLC (*G+fulfillment_deadline*-1) blocks before the HTLC expiry in the current channel in order to guarantee that the channel can remain off-chain if both parties follow the protocol. This is because if the secret is received by the fulfilling user more than *fulfillment_deadline*, but less than (*G+fulfillment_deadline*), blocks before the HTLC expiry, with the current Lightning protocol the fulfilling user could start to fulfill the HTLC off-chain but not completely update both Commitment transactions by *fulfillment_deadline* blocks before expiry, at which point they could fail the channel and go on-chain.

This inequality is slightly different from the one derived in the BOLT #2 specification, as the analysis here includes 4 (rather than 2) instances of the transaction-propagation-and-mining parameter $S$[16].

### *min_final_cltv_expiry*

The *min_final_cltv_expiry* parameter gives the number of blocks from the current blockheight to the HTLC expiry in the last hop. This parameter requires delays corresponding to steps 4, 5 and 6 in the *cltv_expiry_delta* analysis above, so it should be set to $G + 2L = G + 2R + 2S$.

### *fulfillment_deadline*

The *fulfillment_deadline* parameter gives the number of blocks before the HTLC expiry when the party receiving the HTLC goes on-chain[17] unless both parties' Commitment transactions are updated to reflect that the HTLC has been fulfilled. This parameter requires delays corresponding to steps 5 and 6 in the *cltv_expiry_delta* analysis above, so it should be set to $2L = 2R + 2S$[18].

### *to_self_delay*

The *to_self_delay* parameter gives the relative delay before the other party in a channel can receive the payout from a transaction that the other party has put on-chain. This parameter is required to allow the current party to put a conflicting transaction on-chain (thus preventing payout to the other party) in the case where the other party's transaction violates the channel protocol (e.g., it is for an old state). In the worst case:

1. The current party first detects the other party's transaction when it is $R$+1 blocks deep in the current party's version of the blockchain.
2. The current party then experiences $U$ unintentional unavailability before they are able to submit their conflicting transaction.
3. This conflicting transaction then takes $L$ blocks before it is fixed, thus preventing the other party from performing their attempted spend.

The above analysis establishes the following inequality:

$$to\_self\_delay \geq U + L + R + 1 = U + 2R + S + 1.$$

## A.3 Timing Parameters for the WF Protocol

The timing parameters for the WF channel protocol are specified in terms of the timing model presented above.

16 It is unclear why no propagation and mining time for steps 3 and 5 are required in the BOLT #2 specification. This same issue affects the definitions of the *min_final_cltv_expiry* and *fulfillment_deadline* parameters.

17 Assuming that party has fulfilled the HTLC.

18 The value required in the BOLT #2 specification is $2R + G + S$. It is unclear why a grace period is included in that specification, as no grace period is required from the time a party fails the channel and goes on-chain to the expiry of the HTLC from which that party is receiving payment.

DLUs set their *timeout_deadline*, *cltv_expiry_delta*, *min_final_cltv_expiry* and *to_self_delay* parameters as specified in Appendix A.2. DLUs do not have a *fulfillment_deadline* parameter, as they do not go on-chain to fulfill an HTLC offered to them based on the expiry of the HTLC.

CLUs set their:

- *timeout_deadline* and *fulfillment_deadline* parameters as specified in Appendix A.2,
- *min_final_cltv_expiry* parameter to $B + G + 2L$, which is $B$ greater than the value set by DLUs in order to allow the CLU to obtain the current version of the blockchain following a period of intentional unavailability[19],
- $I_S$ parameter to $B + G$, which allows the CLU to obtain the current version of the blockchain, attempt to update the channel state off-chain (if required), and submit one or more transactions to the blockchain (if required),
- $I_L$ parameter to their desired value (subject to the constraint that $I_L \geq I_S$), and
- *to_self_delay* parameter to $I_L + U + L + R + 1 = I_L + U + 2R + S + 1$, which is $I_L$ greater than the value set by DLUs, thus enabling the CLU to be intentionally unavailable for up to $I_L - I_S$ time, obtain the current version of the blockchain, and then respond to an on-chain transaction that violates the protocol as a DLU would respond.

CLUs do not route payments, so they do not specify a *cltv_expiry_delta* parameter.

19 If one-shot receives are supported, the CLU can set their *min_final_cltv_expiry* parameter to $G + 2L$, as it is not necessary to check the blockchain in order to receive a payment.