Factory-Optimized Channel Protocols For Lightning

John Law

September 19, 2022

Version 1.0

Abstract

Layer 2 protocols, such as the Lightning Network, improve Bitcoin's scalability by allowing users to make bitcoin payments without putting transactions on-chain. In addition, factories have been proposed that allow multiple two-party channels to be created with a single on-chain transaction, further improving scalability. The Lightning Network uses Hash Time-Locked Contracts (HTLCs) to implement payments across multiple channels. If one of the parties to an HTLC is unresponsive, the other party must resolve the HTLC on-chain. This creates two problems when using factories. First, with existing channel protocols, the latency to close the factory must be included in the HTLC, resulting in the slow resolution of payment disputes and the stranding of channel capital. Second, the factory must be closed to resolve the HTLC, thus limiting the scalability achieved. This paper presents factory-optimized channel protocols that solve both of these problems. In addition, it gives a factory-optimized channel protocol that supports watchtower-freedom and one-shot receives for casual users. No change to the underlying Bitcoin protocol is required.

1 Overview

While the Lightning Network **[AOP21][BOLT][PD16]** greatly improves Bitcoin's scalability, factories that allow multiple two-party channels to be created and closed with a small number of on-chain transactions are essential if Bitcoin is to be widely used in a trust-free manner **[BDW18]**. Unfortunately, the existing channel protocols are not optimized for use within factories, thus limiting the efficiency of both the channels and the factories. Specifically, the Lightning Network uses Hash Time-Locked Contracts (HTLCs) **[BWHTLC]** to implement payments across multiple channels. If one of the parties to an HTLC is unresponsive, the other party must resolve the HTLC on-chain. This creates two problems.

First, the HTLC's expiry must be delayed by the time required to close the factory and put the channel containing the HTLC on-chain. The most efficient known factory **[BDW18]** can be closed unilaterally in *O*(log *S*) time using *O*(log *S*) on-chain transactions, assuming the factory supports *S* states[1]. If all the hops in a multi-hop payment use channels that are implemented with factories, the sum of the delays for closing all of those factories must be included in the HTLC expiry of the first hop. As a result, this delay could become very large, thus leading to inefficient use of the channels' capital and long waits to obtain payment receipts.

Second, the requirement to close a factory due to the need to resolve an HTLC on-chain means that a single unresponsive party can force the closure of an entire factory, thus limiting the factory's ability to scale Bitcoin.

This paper presents factory-optimized channel protocols that solve both of these problems. The first protocol, called the Partially-Factory-Optimized (PFO) protocol, solves the first problem, while the second protocol, called the Fully-Factory-Optimized (FFO) protocol, solves both problems. Both protocols are slight modifications of the Tunable-Penalties (TP) protocol **[Law22b]** and they share many of its properties, including:

- tunable penalties for putting old transactions on-chain, and
- efficient watchtowers with O(log S) storage for supporting O(S) channel states.

In addition, a version of the FFO protocol, called the Fully-Factory-Optimized-Watchtower-Free (FFO-WF) protocol, is presented that supports watchtower-freedom and one-shot receives for casual users **[Law22a]**. No change to the underlying Bitcoin protocol is required for any of these protocols.

The rest of this paper is organized as follows. The PFO and FFO protocols are presented in Sections 2 and 3, respectively. Section 4 gives the FFO-WF protocol. Sections 5 and 6 present related work and conclusions. An analysis of the most efficient known factory **[BDW18]** is given in Appendix A. Proofs of correctness and the setting of timing parameters are given in Appendices B and C.

2 The Partially-Factory-Optimized (PFO) Protocol

2.1 Overview

The PFO protocol is a slight modification of the TP protocol **[Law22b]**. The TP protocol, in turn, is based on the Lightning channel protocol, but it uses separate value and control transactions. Specifically, each party using the TP protocol has their own on-chain Individual transaction, the output of which they spend with their State transaction. This State transaction is a control transaction that establishes the channel's state and has an HTLC control output corresponding to each HTLC outstanding in the channel in that state. Each HTLC control output is used to resolve an HTLC by spending the output with either an HTLC-success or an HTLC-timeout transaction. Critically, the State, HTLC-success and HTLC-timeout transactions can be put on-chain without spending any of the

[1] See Appendix A for an analysis of how the number of on-chain transactions and the factory close time can be traded-off.

channel's funds. As a result, the TP protocol almost solves the problem of resolving a channel's HTLCs without waiting for the channel's Funding transaction to be put on-chain.

Unfortunately, there is one problem in trying to use the TP protocol to resolve the HTLCs of a factory-created channel before the factory is closed and its Funding transaction is on-chain. The correctness of the TP protocol depends on the ability to put the current Commitment transaction (spending the output of the channel's Funding transaction) on-chain as soon as possible once the relative delay between it and its corresponding State transaction has been met. This relative delay is set to *tsdAB*, which is the maximum of the two parties' *to_self_delay* parameters. The problem is that the latency to close the factory and put the channel's Funding transaction on-chain could exceed *tsdAB*. As a result, the TP protocol cannot be used in such a factory.

The PFO protocol fixes this by simply setting the relative delay between the State transaction and its associated Commitment transaction to the maximum of the factory-close latency and *tsdAB*.

2.2 Protocol Specification

The PFO protocol for a channel shared by Alice and Bob is shown in Figure 1. This figure shows a state *i* in which Alice has an HTLC offered to Bob for a multi-hop payment, with the payment's next hop being in a channel shared by Bob and Carol.

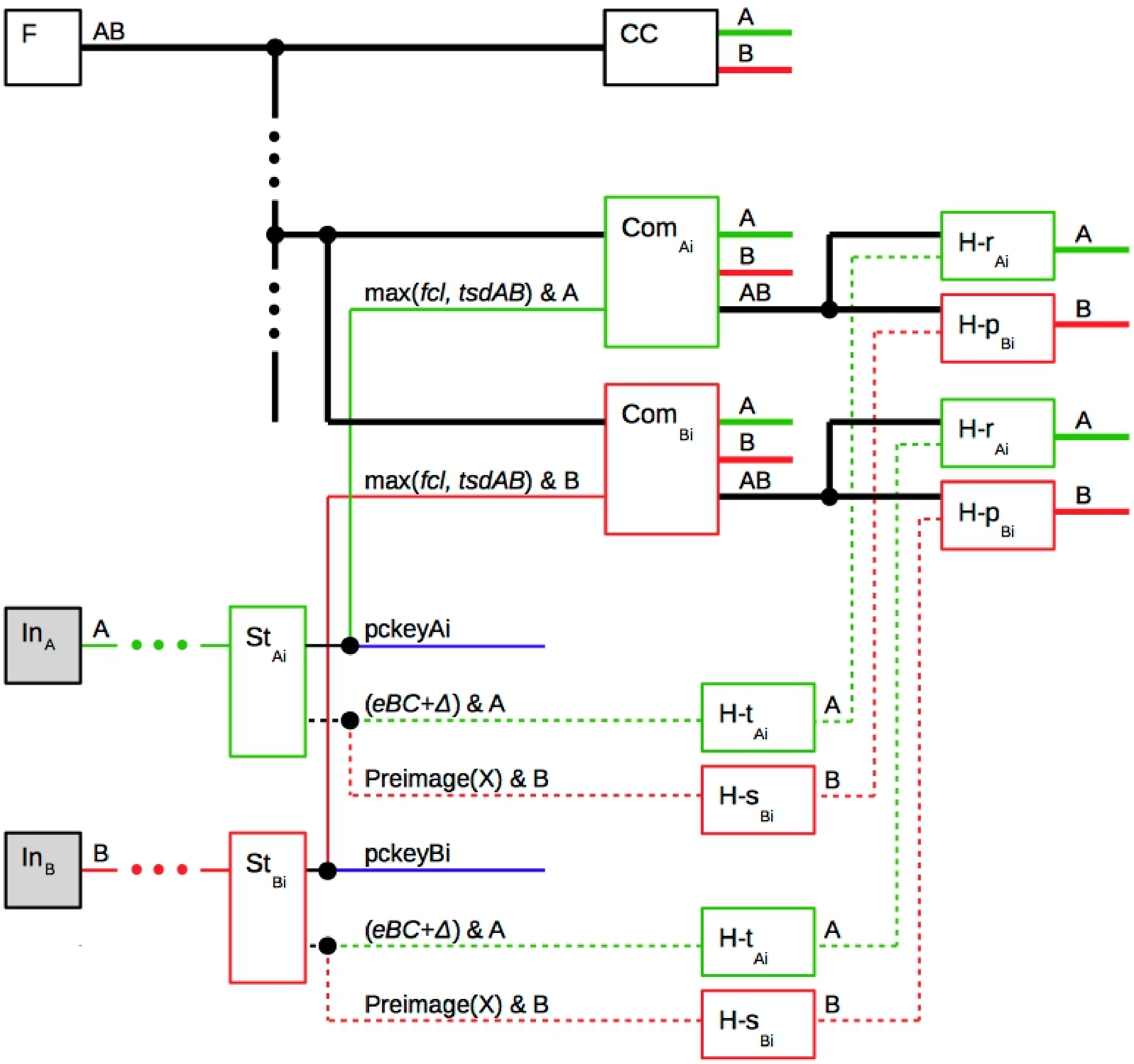

Figure 1. The PFO Channel Protocol. The relative delay from the State (St) transactions to the Commitment (Com) transactions is increased to cover the factory-close latency.

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,
- **pckey{A|B}i** denotes a signature using a per-commitment key for revoking {Alice's|Bob's} state *i* transaction,

- ***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,
- ***fcl*** denotes the latency required to close the factory and put the Funding transaction on-chain,
- ***tsdAB*** denotes the maximum of the *to_self_delay* channel parameters set by Alice and Bob, and
- **Preimage(X)** denotes the payment secret (and payment receipt) which is the preimage of X.

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

- **F** for the Funding transaction,
- **CC** for a Cooperative Close transaction,
- **In** for an Individual transaction,
- **St** for a State transaction,
- **Com** for a Commitment transaction,
- **H-t** for an HTLC-timeout transaction,
- **H-s** for an HTLC-success transaction,
- **H-r** for an HTLC-refund transaction, and
- **H-p** for an HTLC-payment 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.

Bold lines carry channel funds, thin solid lines have value equal to, or slightly larger than[2], the tunable penalty amount, and dashed lines have the minimal allowed value (as they are used for control). When a single output can be spent by multiple off-chain transactions, those transactions are said to *conflict*, and only one of them can be put on-chain. A party will be said to *submit* a transaction when they attempt to put it on-chain.

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

The operation of the PFO protocol matches that of the TP protocol **[Law22b]**. Because the relative delays between both parties' State transaction and their corresponding Commitment transaction are equal, the correctness proof for the TP protocol **[Law22b]** also applies to the PFO protocol.

3 The Fully-Factory-Optimized (FFO) Protocol

3.1 Overview

While the PFO protocol separates the latency to close the factory from the setting of the HTLCs' expiries, it still requires that the factory be closed in order to guarantee that the HTLCs have been resolved correctly. In particular, the PFO protocol uses the race between the parties' conflicting Commitment transactions to guarantee that the State transaction that resolves the HTLCs was put on-chain early relative to the HTLCs' expiries, and the Commitment transactions cannot be put on-chain without closing the factory. In contrast, the FFO protocol makes several changes in order to resolve HTLCs without requiring the closure of the factory.

Consider an HTLC offered by Alice to Bob.

First, only Bob's State transaction has an HTLC control output that determines the resolution of this HTLC, regardless of which party's Commitment transaction is put on-chain. As a result, there is no race to put one's Commitment transaction on-chain, and thus no need to close the factory in order to resolve HTLCs. As another result, it eliminates the possibility of this HTLC being resolved with Alice's State and associated HTLC-timeout transactions being put on-chain late relative to the HTLC's expiry.

Second, because the HTLC is always resolved based on an HTLC control output in Bob's State transaction, Bob has to be incentivized to put his correct State transaction on-chain (or else he could prevent the HTLC from timing out by not putting his State transaction on-chain). This is solved by requiring Bob's State and HTLC-success transactions in order to pay the HTLC, and to refund the HTLC to Alice (after a suitable relative delay) if Bob's State and HTLC-success transactions are not on-chain.

Third, Bob is prevented from putting his State transaction on-chain late relative to the HTLC's expiry by adding a relative delay of *tsdA* (Alice's *to_self_delay* parameter) before he can put his HTLC-success transaction on-chain. This guarantees that Alice will have time to respond with a conflicting transaction[3] that prevents Bob's HTLC-success transaction from being put on-chain late relative to the HTLC's expiry.

Finally, if the HTLC's secret were not revealed until the HTLC-success transaction is put on-chain, the worst-case latency for obtaining a secret from a successful HTLC would depend on *tsdA*, which would greatly increase Bob's *cltv_expiry_delta* parameter, which in turn would increase the cost of capital

[3] Rather than allowing this conflicting transaction to be put on-chain immediately upon the HTLC's expiry, this conflicting transaction cannot be put on-chain until *tsdA* after the HTLC's expiry, thus matching the HTLC-success transaction's relative delay of *tsdA*.

reserved for the HTLC and the delay for obtaining a payment receipt. This problem is solved by introducing a new transaction, called an *HTLC-kickoff* transaction, that spends the HTLC control output in Bob's State transaction and reveals the HTLC's secret, with the HTLC-success transaction spending the HTLC-kickoff transaction's output. Thus, the revelation of the HTLC's secret is performed first, followed by the resolution of the HTLC approximately *tsdA* later[4].

3.2 Protocol Specification

The FFO protocol for a channel shared by Alice and Bob is shown in Figure 2, which shows a state *i* in which Alice has an HTLC offered to Bob.

[4] This technique is similar to the current Lightning protocol's use of an HTLC-success transaction to reveal the HTLC's secret before waiting *tsdA* to allow the HTLC-success transaction to be revoked if it is for an old state **[BOLT]**.

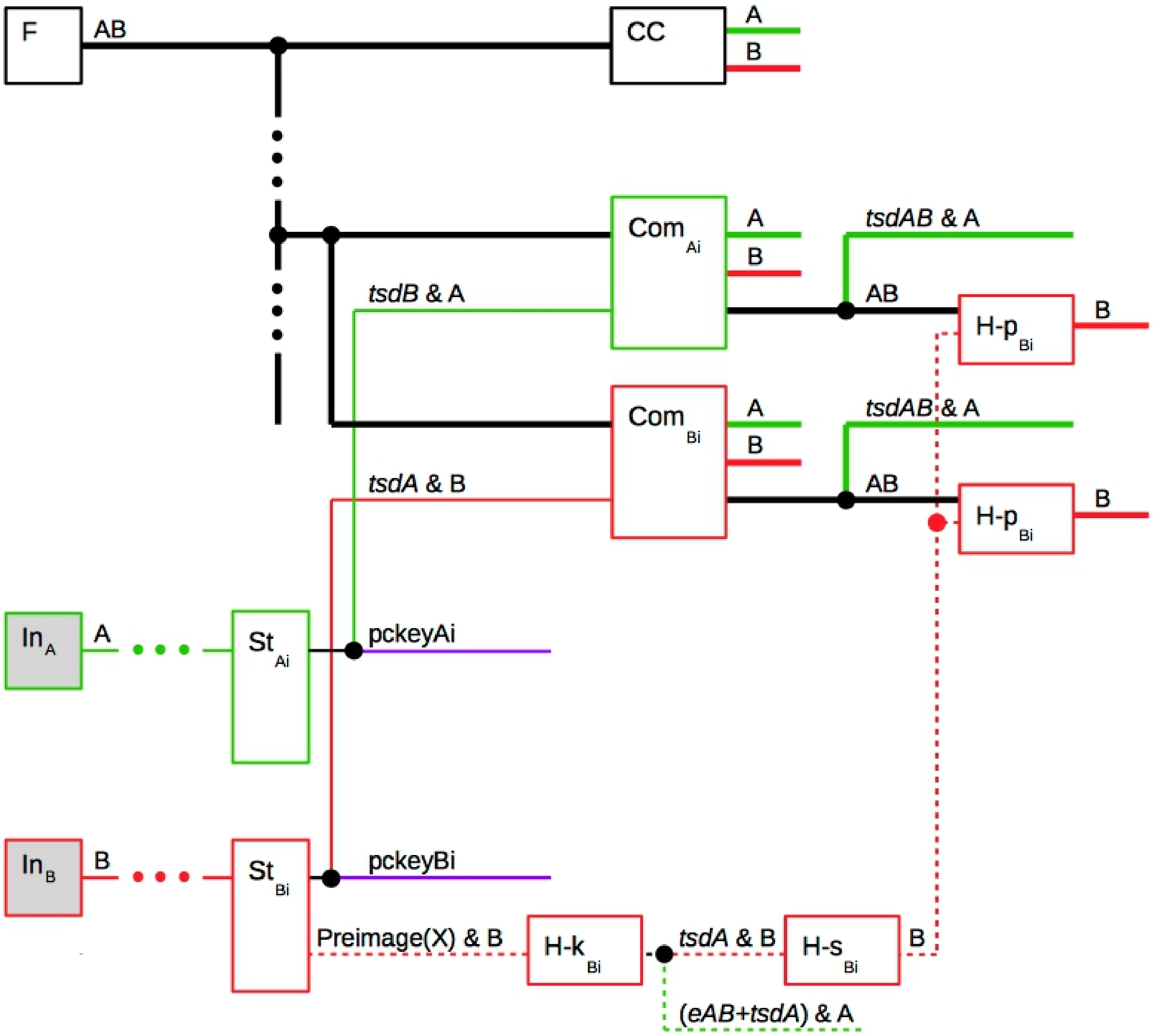

Figure 2. The FFO Channel Protocol. Bob's HTLC-kickoff (H-k) and HTLC-success (H-s) transactions are required for him to receive the payment of the HTLC offered to him. The race between Bob's HTLC-success transaction and Alice's conflicting transaction is delayed by *tsdA* relative to the HTLC's expiry, but the HTLC's secret is revealed earlier by his HTLC-kickoff transaction.

Bob's HTLC-kickoff (H-k) transaction is added to those in Figure 1, where:

- ***eAB*** denotes the expiry for this HTLC in this channel, and

- ***tsd{A|B}*** denotes {Alice's|Bob's} *to_self_delay* channel parameter.

When Bob learns Preimage(X), he sends Preimage(X) to Alice and attempts to update the channel state off-chain to reflect payment of the HTLC. If Bob is unable to update the channel state off-chain, he puts his State and HTLC-kickoff transactions on-chain, waits approximately *tsdA*, and then submits his HTLC-success transaction as soon as it has met its relative delay requirement.

Alice monitors the blockchain and learns Preimage(X) if she detects Bob's HTLC-kickoff transaction. If Alice does not receive Preimage(X) by *eAB* (the HTLC's expiry), at *eAB+tsdA* she checks the blockchain for Bob's HTLC-kickoff transaction. If it is on-chain, Alice submits a transaction that spends the HTLC-kickoff transaction's output. If it is not on-chain, Alice monitors the blockchain and attempts to spend the HTLC-kickoff transaction's output whenever she detects it on-chain.

Both parties also monitor the blockchain for the other party's State transaction, and if they detect an old State transaction on-chain, they revoke it by using the corresponding per-commitment key to spend its first output. Once either party's Commitment transaction is on-chain, each party attempts to spend its HTLC outputs using HTLC-payment transactions for those HTLCs for which they have HTLC-success transactions on-chain. Once a Commitment transaction has been on-chain for *tsdAB*, the party that offered each HTLC attempts to spend the Commitment transaction's corresponding HTLC output if it has not already been spent.

In order to create a new state, parties share signatures for the new state's HTLC-payment and Commitment transactions (in that order), and then revoke their transactions for the previous state by sharing the previous state's per-commitment key.

While Figure 2 only shows both parties' transactions for the same state *i*, it is possible that one or both parties has multiple current State transactions that have not been revoked. This is because the off-chain protocol for changing the channel's state first gives a party signatures for transactions for the new state and then revokes the transactions for the previous state. As a result, before those transactions are revoked, a party can have current (unrevoked) transactions for two states. Therefore, it is necessary to also create HTLC-payment transactions for all pairs of current states.

In addition, care must be taken in the order in which state updates for a successful HTLC are made. Consider an HTLC offered by Alice to Bob in state *i* for which Bob shared the secret with Alice prior to the HTLC's expiry. Alice must create Commitment and State transactions for state *i*+1 that reflect the payment of the HTLC and she must revoke her state *i* transactions before Bob revokes his state *i* transactions. Otherwise, Alice could put her state *i* Commitment transaction, with its HTLC output, on-chain, while Bob could no longer put an HTLC-success transaction on-chain in order to force payment of the HTLC.

Appendix B specifies the parties' timing parameters and uses them to prove the correctness of the FFO protocol.

3.3 Extensions

Unilateral Close after an Old Transaction is Put On-Chain

If a party accidentally puts on old State transaction on-chain, they only lose the penalty amount that is the output of that transaction (and potentially some of the minimal values of that transaction's HTLC control outputs). However, once their State transaction has been revoked, they have lost the ability to force a unilateral close of the channel after the Funding transaction is on-chain.

To address this, it is possible to add a Trigger (or Kickoff **[BDW18]**) transaction that spends the output of the Funding transaction, as was described for the TP protocol **[Law22b]**. After the Trigger transaction has been on-chain for 3*tsdAB* (in order to allow the other party to put their Commitment transaction on-chain), the Decker-Wattenhofer protocol **[DW15]** can be used to settle the channel.

Off-Chain Control Outputs

As was the case for the TP protocol **[Law22b]**, a party that operates multiple channels using the FFO protocol can use a single UTXO to fund the inputs to the State transactions in all of their channels.

Continuing to Operate the Channel

The protocol presented above shows how a channel's HTLCs can be resolved on-chain without putting its Funding transaction on-chain, and thus without closing the channel factory. The party that went on-chain can settle the channel and its HTLCs correctly, even if their partner is malicious. However, the most likely reason a party goes on-chain is that their partner is unintentionally unavailable, rather than malicious. When the unavailable partner becomes available again, the two parties could choose to re-start operating the channel, and that is possible because the channel's funds have not been distributed.

For example, assume Alice was unresponsive and Bob put his State and associated HTLC-kickoff and HTLC-success transactions on-chain. When Alice becomes available, she and Bob can update the channel state off-chain to reflect the resolution of the HTLCs as was established by Bob. Of course, the output of Bob's Individual transaction has already been spent by his State transaction, so he needs a new UTXO to use for his new channel State transactions. Any UTXO owned by Bob could be agreed upon, or Alice and Bob could agree to use the unspent first output of Bob's State transaction as playing the role of his new Individual transaction's output. This output has value equal to the penalty amount, which is slightly smaller than would be required to provide both a penalty amount and funds for HTLC control outputs for a new State transaction. However, it seems reasonable to allow Bob to provide a slightly smaller penalty amount going forward, given that it was Alice's unavailability that forced him to go on-chain.

4 Watchtower-Freedom for Casual Users

4.1 Overview

A recent paper introduced the Watchtower-Free (WF) protocol for Casual-Lightning-Users (CLUs) who do not want to use a watchtower **[Law22a]** or route Lightning payments for others. That paper also showed how to support one-shot receives for CLUs, thus simplifying their reception of payments. The same techniques can be applied to the FFO protocol to obtain a protocol that maintains its factory optimizations while supporting watchtower-freedom and one-shot receives for CLUs, although several problems have to be addressed in doing so. The resulting protocol will be called the FFO-WF protocol.

Let Alice be a CLU who is sending a payment and let Bob be her Dedicated-Lightning-User (DLU) channel partner. In order to implement the WF or FFO-WF protocol **[Law22a]**, Alice specifies 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).

Alice 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.

First-Hop Protocol

Before describing the FFO-WF protocol's operation, it is helpful to review how CLUs send payments using the WF protocol. The WF protocol allows Alice to send a payment without Bob going on-chain by adding a relative delay of *tsdB* (Bob's *to_self_delay* parameter) before Alice can spend the HTLC output for her payment from either party's Commitment transaction. As a result, Bob can remain off-chain even after the expiry of the HTLC, while Alice can force Bob to provide a payment receipt (or to not take the payment funds from Alice) by putting her Commitment transaction on-chain.

In contrast, in the FFO protocol only Bob's State transaction has an HTLC control output for Alice's payment. As a result, if the protocol were modified to allow Bob to keep his State, HTLC-kickoff and HTLC-success transactions off-chain past the HTLC's expiry, Alice would not have any means to force Bob to provide a payment receipt (as putting her State transaction on-chain would not do so). Therefore, in the FFO-WF protocol, Alice's (rather than Bob's) State transaction has an HTLC control output for each HTLC offered by Alice. This HTLC control output can be spent immediately by Bob's HTLC-success transaction (thus revealing the HTLC's secret) or by Alice after both a relative delay of *tsdB* and an absolute delay of the HTLC's expiry. Alice can force Bob to produce a receipt (or not receive payment) by putting her State transaction on-chain and then attempting to spend its HTLC control output approximately *tsdB* later.

In addition, Bob needs to ensure that he receives the HTLC's payment if he provides its secret to Alice early enough. As long as Alice eventually puts her State transaction on-chain, Bob can force Alice to pay the HTLC if he knows its secret sufficiently early relative to its expiry:

- If Alice puts her State transaction on-chain at least *tsdB* before the HTLC's expiry, Bob will be able to put his HTLC-success transaction on-chain before the expiry, and thus before Alice's conflicting transaction.
- If Alice puts her State transaction on-chain later than *tsdB* before the HTLC's expiry, the *tsdB* relative delay before Alice can put her conflicting transaction on-chain guarantees that Bob will be able to put his HTLC-success transaction on-chain.

Finally, Bob needs a way to force Alice to eventually put her State transaction on-chain. The solution is to design Bob's Commitment transaction to always pay the funds from all outstanding HTLCs (both those offered to Bob and those offered by Bob) to Bob, but to delay Bob's State and Commitment transactions enough that Alice can put her State and Commitment transactions on-chain instead. Because Bob's Commitment transaction resolves all HTLCs in Bob's favor, Alice is incentivized to put her State and Commitment transactions on-chain instead.

The FFO-WF first-hop protocol with a single payment outstanding from Alice is shown in Figure 3[5].

[5] Figure 3 shows the version of the FFO-WF protocol that uses a single-input State transaction and thus supports only 2^{24} channel states. It would be possible to add a second input to Bob's State transaction to support 2^{48} channel states, as described in **[Law22a]**.

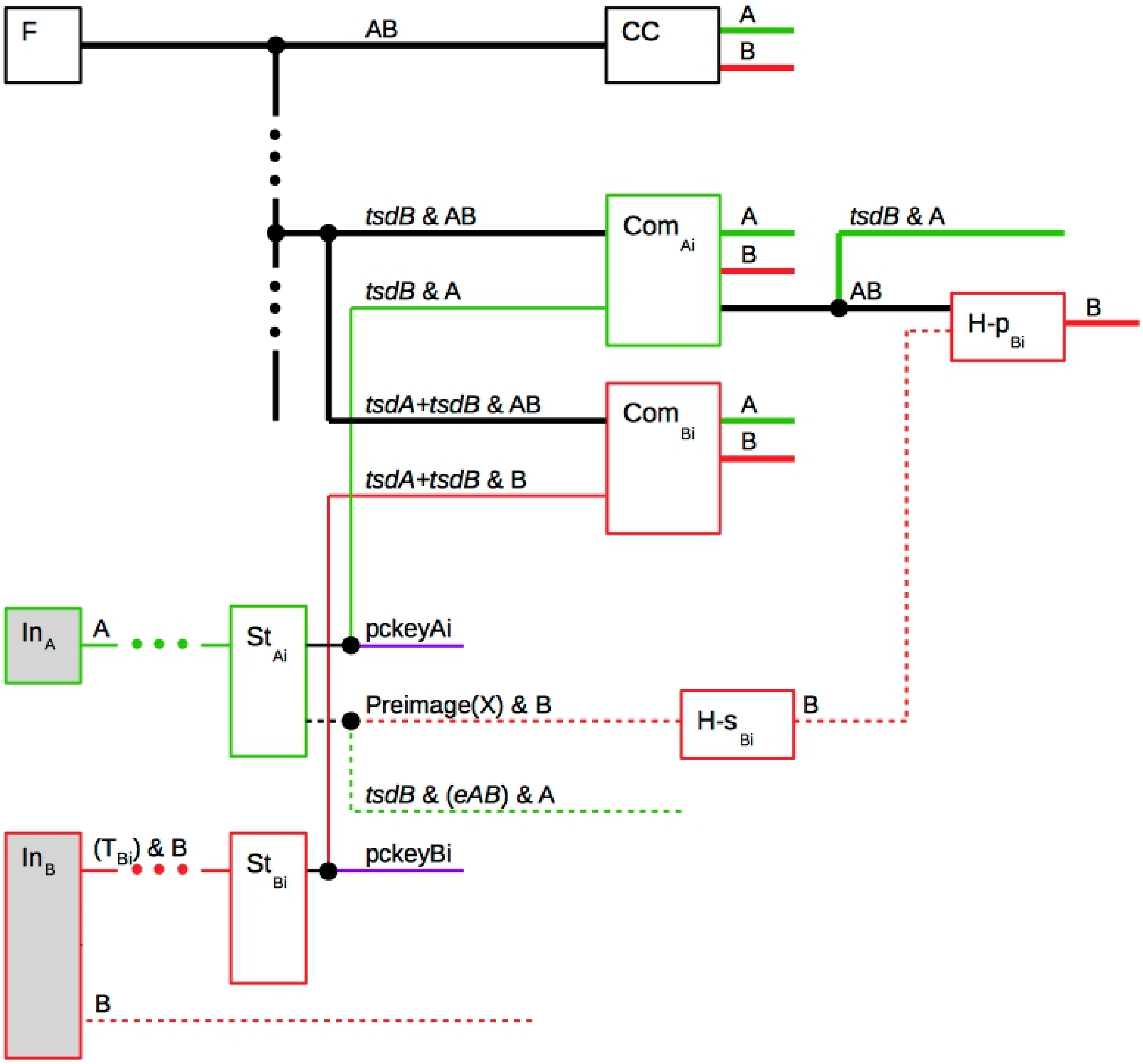

Figure 3. The FFO-WF first-hop protocol. CLU Alice has one HTLC outstanding to DLU Bob in state *i*. Alice's Commitment transaction has outputs for all outstanding HTLCs, while Bob's Commitment transaction has no HTLC outputs and pays all HTLCs to Bob. Bob's State transaction has an absolute delay until time T_{Bi} that allows Alice to put her Commitment transaction on-chain if desired. The second output in Bob's Individual transaction is unused here, as it is only used in the last-hop protocol.

In Figure 3 (and the rest of the paper):

- **(*T_{Bi}*)** denotes an absolute timelock in Bob's State *i* transaction that is set to *tsdA* in the future when Alice signs Bob's corresponding Commitment transaction.

Last-Hop Protocol

In order to allow Alice to make one-shot receives **[Law22a]**, the FFO-WF protocol includes an HTLC control output in Alice's State transaction for each HTLC that is offered to her. Alice can spend this HTLC control output with her HTLC-success transaction that reveals the HTLC's secret. Because the process of receiving a payment is one-shot for Alice, there cannot be a relative delay between her State transaction and her HTLC-success transaction. As a result, there is a risk that Alice could put both her State and HTLC-success transactions on-chain simultaneously and far after the HTLC's expiry. This is prevented by forcing Alice's HTLC-success transaction for state *i* to also spend an HTLC control output in Bob's Individual transaction which can also be spent by Bob's HTLC-timeout transaction for state *i*. Because Bob's HTLC-timeout transaction can be put on-chain after the HTLC's expiry, Bob is able to prevent Alice from putting her HTLC-success transaction on-chain late relative to the HTLC's expiry.

Finally, there is the risk of Bob using an old HTLC-timeout transaction, or any transaction other than the current state *i* HTLC-timeout transaction (as the HTLC control output in Bob's Individual transaction requires only Bob's signature), to prevent Alice's HTLC-success transaction from being put on-chain. This risk is eliminated by having the HTLC output in Alice's state *i* Commitment transaction pay the HTLC output to Alice (after a suitable relative delay) unless Bob's state *i* HTLC-timeout transaction is on-chain. As a result, if Bob spends the HTLC control output in his Individual transaction with anything other than his state *i* HTLC-timeout transaction, Alice will be able to receive the HTLC payment.

The FFO-WF first-hop protocol with a single payment outstanding from Alice is shown in Figure 4.

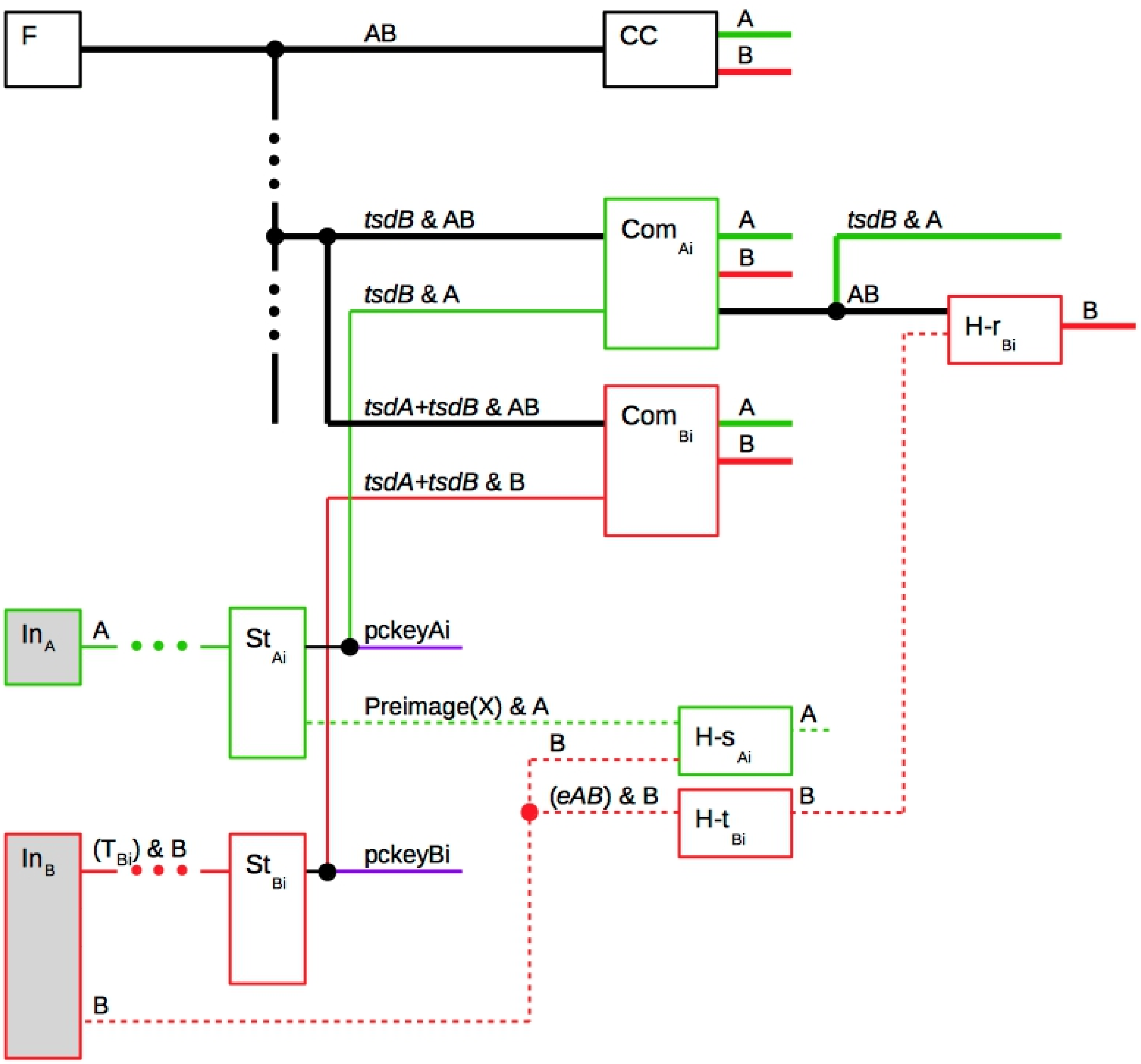

Figure 4. The FFO-WF last-hop protocol. In state *i*, Bob has one outstanding HTLC offered to Alice. Alice's HTLC-success transaction and Bob's HTLC-timeout transaction conflict as both spend an HTLC control output in Bob's Individual transaction. Alice can receive the payment for the HTLC unless Bob puts his state *i* HTLC-timeout transaction on-chain after the HTLC's expiry. Alice can prevent Bob from doing so by putting her HTLC-success transaction on-chain, thus revealing its secret.

Bob's HTLC-timeout transactions do not spend an output of a State transaction, so his HTLC-timeout transaction for each state must be made unique in some manner. This can be accomplished by including state number information in its nSequence field (as in the Lightning protocol), by including state number information in its output's locking script, or any other means[6].

[6] The *eAB* absolute delay in Bob's HTLC-timeout transaction is implemented in its nLocktime field. There is no CHECKLOCKTIMEVERIFY operator in his Individual transaction's locking script for the second output.

4.2 Protocol Specification

Updating the Channel State

In order to establish a new channel state, both parties:

1. calculate the State, HTLC-timeout, HTLC-success, Commitment, HTLC-refund and HTLC-payment transactions for the new state[7],
2. exchange partial signatures for the new state's HTLC-refund, (Bob's) HTLC-payment, Commitment and (Alice's) HTLC-success transactions (in that order)[8], and
3. exchange per-commitment keys for the old state, thus revoking it.

In addition, the following constraint is added in order to guarantee one-shot receives:

- Whenever state *i* includes an HTLC offered to Alice, state *i*+1 does not include any HTLCs offered to Alice.

This constraint is required because the process of creating a new state off-chain is not atomic, and thus Alice can have signed and unrevoked State, HTLC-success and Commitment transactions for both state *i* and state *i*+1. If states *i* and *i*+1 both had HTLC control outputs for HTLCs offered to Alice, Bob would not know whether to submit his state *i* or his state *i*+1 HTLC-timeout transaction for an HTLC that is past its expiry, and whichever one he chose, Alice could then put her transactions for the other state on-chain, thus receiving payment for the HTLC despite revealing its secret far after its expiry.

This constraint can always be met by having Alice reveal the secrets for all outstanding HTLCs offered to her (as she is the destination for the payments corresponding to those HTLCs, so she knows their secrets) and thus removing those HTLCs from the next channel state. Once a channel state without HTLCs offered to Alice has been created, the following state can once again include HTLCs offered to her.

Channel Maintenance

At any time, and at least once every I_L, Alice:

- updates the channel state off-chain with Bob to reflect a new pre-payment for Bob's cost of capital in the channel, and
- checks the blockchain to see if Bob has put a State transaction on-chain, and if she detects such a transaction:

[7] This step requires that each party shares their per-commitment pubkey for the new state with the other party.

[8] Specifically, each party sends their partial signature for every transaction input that requires it, but only if the transaction can be put on-chain by the other party. Thus, Bob provides his signature for the second input in Alice's HTLC-success transaction, but Alice does not provide a signature for Bob's HTLC-success transaction, as it is not required.

  - if it is old, she revokes it by using a per-commitment key to spend its first output (thus taking the penalty amount for herself), or
  - if it is current, she puts the channel state on-chain (as described below)[9].

If Alice spends a grace period of length G[10] attempting to update the channel state without success, she puts the channel state on-chain (as described below).

At all times, Bob:

- cooperates with Alice to update the channel state off-chain,
- submits each HTLC-timeout transaction for Alice's current state once the expiry of that HTLC is reached,
- checks the blockchain to see if Alice has put a State transaction on-chain, and if he detects such a transaction:
  - if it is old, he revokes it by using a per-commitment key to spend its first output (thus taking the penalty amount for himself), or
  - if it is current, he submits each HTLC-success transaction that spends an output from her State transaction as soon as he knows that HTLC's secret, and he puts the channel state on-chain (as described below), and
- checks the blockchain to see if Alice has put her current Commitment transaction on-chain, and if he detects it he:
  - submits each HTLC-payment transaction for which his corresponding HTLC-success transaction is on-chain.

In addition, if Alice has not updated the channel state within I_L to reflect a new pre-payment for Bob's cost of capital, Bob puts the channel state on-chain as soon as possible (as described below).

Putting the Channel State On-Chain

If either party wants to close the channel, they can try to obtain a signed Cooperative Close transaction. However, if they do not receive a signed Cooperative Close transaction quickly enough, they can put the channel state on-chain themselves.

In order to put the channel state on-chain, Alice submits her current State transaction, along with any associated HTLC-success transactions and any associated transactions that conflict with Bob's HTLC-success transactions (when their relative delay and absolute timelock are reached). Then, Alice checks the blockchain at least once every I_L and:

[9] This case will never occur if Alice follows the protocol, but it is included in case she has unintentional unavailability that prevents her from following the protocol.

[10] The parameter G matches the parameter G of the Lightning protocol **[BOLT]** and the WF protocol **[Law22a]**.

- if her current Commitment transaction is not on-chain and its parents have been on-chain for at least *tsdB*, Alice submits her current Commitment transaction, and
- if her current Commitment transaction has been on-chain for at least *tsdB*, Alice attempts to spend any of its unspent HTLC outputs.

Similarly, in order to put the channel state on-chain, Bob submits his current State transaction to the blockchain as soon as possible[11]. Then, Bob monitors the blockchain and:

- if his current Commitment transaction is not on-chain and its parents have been on-chain for at least *tsdA* + *tsdB*, Bob submits his current Commitment transaction.

Sending a Payment

The procedure for sending a payment follows that of the WF protocol. Once Bob obtains the secret for the payment, he shares the secret with Alice and tries to update the channel state off-chain to reflect the success of the HTLC. If he is unable to update the channel state off-chain due to Alice's unavailability, he stays off-chain for at least I_L, unless Alice goes on-chain. If Alice does go on-chain, Bob responds by putting the channel state on-chain as described above. Even if the HTLC has expired, he will be able to put his HTLC-success transaction on-chain, due to the relative delay before Alice can put her conflicting transaction on-chain.

Receiving a Payment

The procedure for sending a payment follows that of the WF protocol. Once Alice has given the secret for the payment to Bob, she tries to update the channel state off-chain to reflect the success of the HTLC. If she is unable to update the channel state off-chain within a grace period of length *G* after she signs Bob's transactions for the current state, she puts the current channel state on-chain as soon as possible.

Getting a Payment Receipt

At any time after the expiry of the HTLC for a payment sent by Alice, if she needs to get a receipt and Bob is uncooperative, Alice can put the current channel state on-chain (as described above). Once her current State transaction has been on-chain for *tsdB*, she can submit her transaction that conflicts with Bob's HTLC-success transaction. Either Alice's conflicting transaction will be on-chain (in which case Alice does not have to make the payment) or Bob's HTLC-success transaction (that provides the payment receipt) will be on-chain. As was the case in the WF protocol, the procedure for getting a payment receipt is not one-shot and may be awkward for casual users. However, it is only required when there is both a payment dispute (or other need to get a receipt quickly) and an uncooperative channel partner.

[11] Note that Bob must wait for the absolute timelock in his current State transaction before putting it on-chain.

Timing Parameters and Proof of Correctness

The timing parameters for the FFO-WF protocol and a proof of its correctness are given in Appendix C.

4.3 Extensions

Unilateral Close after an Old Transaction is Put On-Chain

The same technique that was presented in Section 3.3 can be used to allow a party that put an old state on-chain to still force a unilateral channel close after the Funding transaction is on-chain. The only difference is that the first transaction from the Decker-Wattenhofer protocol should have a relative delay of 4*tsdAB* in order to accommodate the delay in Bob's current State transaction.

Merged Control Outputs

As presented above, the FFO-WF protocol requires a separate HTLC control output in Bob's Individual transaction (and a separate HTLC output in Alice's Commitment transaction) for each HTLC offered to Alice. However, it is possible to use a single HTLC control output in Bob's Individual transaction (and a single HTLC output in Alice's Commitment transaction) that combines all of the HTLCs offered to Alice. This is done by requiring that Alice provide all of the required preimages[12] for all of the HTLCs offered to her in order to put her HTLC-success transaction on-chain, and having the single HTLC output in her Commitment transaction for the HTLC's offered to her provide payment for all of those HTLCs. This optimization is possible because Alice is the destination for all of the payments corresponding to the HTLCs offered to her, so she knows the required preimages for all of them.

Off-Chain Control Outputs

As was the case for the TP protocol **[Law22b]**, a party that operates multiple channels using the FFO-WF protocol can use a single UTXO to fund the inputs to the State transactions in all of their channels.

Continuing to Operate the Channel

The same technique that was presented in Section 3.3 can be used to allow the two parties to re-start channel operation after one of them has been forced to go on-chain with their control transactions due to the other party's unintentional unavailability.

5 Related Work

The protocols presented here are designed to make efficient use of channel factories for Lightning. The concept of creating a channel factory for Lightning, as well as the most efficient published protocol for such a factory, was presented by Burchert, Decker and Wattenhofer **[BDW18]**.

[12] Note that if Point Time Lock Contracts (PTLCs) are used instead, different adaptor signatures corresponding to each of the PTLCs offered to Alice can be used to make her single PTLC-success transaction even more efficient.

Towns proposed adding a new opcode to Bitcoin, called TAPLEAF_UPDATE_VERIFY (TLUV), that would support (among other things) the ability to remove one party from a CoinPool without having to close the CoinPool **[Tow21]**, and ZmnSCPxj noted that this opcode could also be used to remove one channel from a factory without closing the factory **[Zmn22]**. Those proposals differ from the protocols presented here in that they require a change to Bitcoin.

Both the PFO and the FFO protocols use HTLC-timeout and HTLC-success transactions to resolve HTLCs before the channel's Funding transaction has been put on-chain, thus reducing the expiry of those HTLCs. This idea is analogous to, and inspired by, the Lightning protocol's use of HTLC-timeout and HTLC-success transactions to resolve HTLCs before their associated Commitment transaction has been verified to be unrevoked **[BOLT]**.

The PFO and FFO protocols are based on the TP protocol presented by Law **[Law22b]**. The techniques for allowing the FFO protocol to support casual users are based on those presented by Law in **[Law22a]** and **[Law22b]**.

6 Conclusions

This paper presents new channel protocols that are optimized for use within channel factories. The PFO protocol makes the time to resolution of HTLCs within a channel in a factory independent of the latency to close the factory. The FFO protocol achieves the same result and allows the factory (and the channel) to remain open after the resolution of the HTLCs, and the FFO-WF protocol provides watchtower-freedom and one-shot receives for casual users. While the protocols presented here require the users to have on-chain Individual transactions with individually-owned outputs, those transactions in no way depend on a factory or its channels. As a result, as long as a user has a suitable on-chain transaction with the required individually-owned output(s), that user can open and close successive channels within factories without spending those individually-owned outputs.

The FFO and FFO-WF protocols both build on the TP protocol and inherit many of its properties. In particular, if the FFO protocol is used in channels owned by pairs of dedicated users and the FFO-WF protocol is used in channels with a casual user:

- casual users do not require watchtowers,
- casual users can use one-shot receives,
- dedicated users can use watchtowers with logarithmic storage,
- all users can have tunable penalties,
- all channels can have HTLCs with expiries that are independent of the latency of closing the factory that created them, and
- all channels can resolve their HTLCs without closing the factories that created them.

Thus, these protocols solve many of usability, efficiency and scalability issues with the current Lightning protocol.

There are some costs incurred in using the FFO and FFO-WF protocols, including:

- increased cost of capital for dedicated users who share channels with casual users,
- increased delays in unilaterally closing channels,
- increased on-chain footprint in the case of a unilateral channel close, and
- increased protocol complexity.

It is hoped that the advantages of these protocols will warrant their implementation and that channel factories will become commonly-used, thus improving Bitcoin's scalability.

References

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

BDW18 Conrad Burchert, Christian Decker and Roger Wattenhofer. Scalable Funding of Bitcoin Micropayment Channel Networks. In Royal Society Open Science, 20 July 2018. See http://dx.doi.org/10.1098/rsos.180089.

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.

DW15 Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In Proc. 17th Intl. Symposium on Stabilization, Safety, and Security of Distributed Systems, August 2015. pp. 3-18. See https://tik-old.ee.ethz.ch/file/716b955c130e6c703fac336ea17b1670/duplex-micropayment-channels.pdf.

GKP94 Ronald Graham, Donald Knuth and Oren Patashnik. Concrete Mathematics, Second Edition. 1994. Addison-Wesley: Upper Saddle River, NJ.

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

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

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

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.

Sab Stars and bars. See https://en.wikipedia.org/wiki/Stars_and_bars_(combinatorics).

Tow21 AJ Towns. TAPLEAF_UPDATE_VERIFY covenant opcode. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2021-September/019419.html.

Zmn22 ZmnSCPxj. Channel Eviction From Channel Factories By New Covenant Operations. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-February/003479.html.

Appendix A: Analysis of Invalidation Trees

The Burchert-Decker-Wattenhofer factories **[BDW18]** use invalidation trees **[DW15]** to update the factory state and create new channels off-chain. Invalidation trees are defined in terms of two parameters:

- ***n***: the number of time windows during which transactions can be put on-chain, and
- ***d***: the number of transactions put on-chain.

The total number of invalidation trees (and thus factory states) that can be created equals the number of strings of length *d* that use an ordered alphabet with *n*+1 characters, where the elements in the string are nondecreasing **[DW15]**. For example, if *d* = 3 and *n* = 2, ten 3-digit nondecreasing strings using the characters 0..2 are possible:

- 000
- 001
- 002
- 011
- 012
- 022
- 111
- 112
- 122
- 222

Each such sequence of d nondecreasing elements can be represented by an arbitrary sequence of d stars and n bars, where each star represents a character and each bar represents an increase of 1 in the value of the character **[Sab]**. This correspondence for the example of $d = 3$ and $n = 2$ is shown below:

- 000 corresponds to ***||
- 001 corresponds to **|*|
- 002 corresponds to **||*
- 011 corresponds to *|**|
- 012 corresponds to *|*|*
- 022 corresponds to *||**
- 111 corresponds to |***|
- 112 corresponds to |**|*
- 122 corresponds to |*|**
- 222 corresponds to ||***

There are thus a total of (($n + d$) choose d) such sequences.

Setting $n = d$, there are ($2n$ choose n) invalidation trees with n time windows and n on-chain transactions. Because ($2n$ choose n) ≈ 4^n / sqrt(πn) (see Exercise 60, p. 495 of **[GKP94]**), it follows that a factory with S states can be closed unilaterally in $O(\log S)$ time using $O(\log S)$ on-chain transactions.

Alternatively, if $n = O(1)$, there are (($n+d$) choose d) invalidation trees with $O(1)$ time windows and d on-chain transactions. Because (($n+d$) choose d) = $d^{O(1)}$, it follows that a factory with S states can be closed unilaterally in $O(1)$ time using $S^{1/O(1)}$ on-chain transactions.

Finally, if $d = O(1)$, there are (($n+d$) choose d) invalidation trees with n time windows and $O(1)$ on-chain transactions. Because (($n+d$) choose d) = $n^{O(1)}$, it follows that a factory with S states can be closed unilaterally in $S^{1/O(1)}$ time using $O(1)$ on-chain transactions.

Appendix B: Timing Parameters and Proof of Correctness for the FFO Protocol

The timing model from Appendix A of **[Law22a]**, including its parameters R, S, G, B, U and L, can be used to set the timing parameters for the FFO protocol and to prove its correctness.

B.1 Timing Parameters for the FFO Protocol

timeout_deadline

In the FFO protocol, the *timeout_deadline* parameter gives the number of blocks **after** the HTLC expiry when the party offering the HTLC first checks the blockchain for the receiving party's HTLC-kickoff transaction. If the receiving party's HTLC-kickoff transaction is not on-chain, the offering party continues to check for it at least once every *tsdA*, which is the offering party's *to_self_delay* parameter. Whenever the offering party detects the receiving party's HTLC-kickoff transaction, they submit their transaction that conflicts with the receiving party's HTLC-success transaction as soon as possible.

The *timeout_deadline* parameter is set to the same party's *to_self_delay* parameter.

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, it will also be fulfilled in the current hop.

The *cltv_expiry_delta* parameter is set to $3L + G - (R+1)$.

fulfillment_deadline

The *fulfillment_deadline* parameter gives the number of blocks before the HTLC expiry when the party receiving the HTLC either goes on-chain if they have the HTLC's secret and have not updated both parties' Commitment transactions to reflect that the HTLC has been fulfilled, or updates both parties' Commitment transactions to reflect that the HTLC has timed out otherwise.

The *fulfillment_deadline* parameter is set to $2L - (R+1)$.

min_final_cltv_expiry

The *min_final_cltv_expiry* parameter gives the number of blocks from the current block height to the HTLC expiry in the last hop. This parameter needs to support a grace period of length G followed by a *fulfillment_deadline*.

The *min_final_cltv_expiry* parameter is set to $2L + G - (R+1)$.

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.

The *to_self_delay* parameter is set to $U + L + (R+1)$.

max_cltv_expiry

The *max_cltv_expiry* parameter gives the maximum number of blocks in the future for an expiry for an HTLC in a signed State transaction[13].

The *max_cltv_expiry* parameter is set to the minimum of the parties' *to_self_delay* parameters.

B.2 Correctness of the FFO Protocol

The correctness of the FFO protocol will be proven assuming that the channel's Funding transaction may appear on-chain at any time (in order to avoid placing any constraints on when it can appear on-chain) and that each party can force the Funding transaction to be put on-chain eventually (as otherwise they would not be able to guarantee that they receive any channel funds at all). It will also be assumed that the parties do not both sign a Cooperative Close transaction, as the existence of such a transaction implies that both parties agree to close the channel in the given state.

The proof that each party is able to ensure that a current (unrevoked) Commitment transaction is put on-chain matches that of the TP protocol **[Law22b]**.

The correct resolution of HTLCs is more complex. There are two cases.

Resolution of an HTLC at an intermediate hop

Consider the case where Bob is offered an HTLC by Alice and Bob offers a corresponding HTLC to Carol in the payment's next hop. In this case, we must show that if Bob follows the protocol and pays the HTLC in the hop with Carol, he is guaranteed to receive payment for the HTLC in the hop with Alice. Assume for the sake of contradiction that Bob pays the HTLC in the hop with Carol, but does not receive payment for the HTLC in the hop with Alice. Let *cltv_expiry_delta* denote Bob's *cltv_expiry_delta* parameter, let *eAB* denote the expiry of the HTLC in the hop with Alice, let *eBC* denote the expiry of the HTLC in the hop with Carol, and note that $eAB = eBC + cltv\_expiry\_delta = eBC + 3L + G - (R+1)$.

In the hop with Carol, Bob checked the blockchain for Carol's HTLC-kickoff transaction at $eBC + tsdB$ and at least once every *tsdB* afterward until Carol's HTLC-kickoff transaction is detected (which must occur eventually, given that Bob paid the HTLC in the hop with Carol). There are two cases:

- **Case 1:** Carol's HTLC-kickoff transaction was fixed at time $eBC + tsdB$. In this case, Bob submitted his transaction that conflicts with Carol's HTLC-success transaction, so Carol's HTLC-success transaction is fixed by $eBC + tsdB + L$ and is in block $eBC + tsdB + L - (R+1)$ at the latest. Therefore, Carol's HTLC-kickoff transaction is in block $eBC + L - (R+1)$ at the latest, which implies Bob learned the HTLC's secret by $eBC + L$. Bob then shared the HTLC's secret with Alice and attempted to update Alice's and Bob's Commitment transactions to reflect the

[13] This parameter is introduced in the FFO protocol in order to prevent the party receiving an HTLC from putting an HTLC-success transaction on-chain but then being unable to put the corresponding HTLC-payment transaction on-chain.

HTLC's success until eAB - $fulfillment\_delay$ = eBC + 3L + G - (R+1) - 2L + (R+1) = eBC + L + G. As a result, Alice and Bob had at least G time to update their Commitment transactions off-chain, but they failed to do so, given the assumption that Bob did not receive payment for the HTLC. Therefore, Bob submitted his State and HTLC-kickoff transactions at eBC + L + G = eAB - 2L + (R+1), which implies that his HTLC-kickoff transaction is fixed by eAB - L + (R+1) and is in block eAB - L or earlier. Bob then submitted his HTLC-success transaction by eAB - L + $tsdA$, so either his HTLC-success transaction or Alice's conflicting transaction is fixed by eAB + $tsdA$. There are two subcases:

- ○ **Subcase 1a:** Bob's HTLC-success transaction is fixed by eAB + $tsdA$. In this subcase, either a) Alice's or Bob's Commitment transaction was fixed when Bob submitted his HTLC-success transaction, or b) neither Commitment transaction was fixed then. If a), then Bob submitted his HTLC-payment transaction with his HTLC-success transaction by eAB - L + $tsdA$, which implies Alice's conflicting transaction won the race and was fixed by eAB + $tsdA$ and in block eAB + $tsdA$ - (R+1) or earlier. If Alice's Commitment transaction is fixed, then it is in block eAB - (R+1) or earlier, which implies that her State transaction is in block eAB - $tsdB$ - (R+1) or earlier, so she submitted it by eAB - $tsdB$ - 1 (as blockchain reorganizations can only affect the top R blocks), which implies she submitted it before it was signed (due to the setting of Bob's max_cltv_expiry parameter[14]), which is a contradiction.
- ○ **Subcase 1b:** Alice's transaction that conflicts with Bob's HTLC-success transaction is fixed by eAB + $tsdA$. In this subcase, Alice's conflicting transaction is fixed before its minimum absolute locktime, which is a contradiction.

- **Case 2:** Carol's HTLC-kickoff transaction was not fixed at eBC + $tsdB$. In this case, Bob checked the blockchain at least once every $tsdB$ and submitted his transaction that conflicts with Carol's HTLC-success transaction as soon as he saw her HTLC-kickoff transaction. Because of the setting of $tsdB$, it is guaranteed that Bob's conflicting transaction was fixed before Carol's HTLC-success transaction, which is a contradiction.

Resolution of an HTLC at the last hop

Consider the case where Bob is offered an HTLC by Alice in the payment's last hop. This case is identical to the previous case of an HTLC at an intermediate hop, except:

- the analysis starts when Bob reveals the HTLC's secret to Alice (which corresponds to eBC + L in the previous analysis), and
- it is based on Bob's *min_final_cltv_expiry* parameter rather than his *cltv_expiry_delta* parameter.

[14] To see this, let T denote when Alice's State transaction was signed. It follows from Bob's max_cltv_expiry parameter that $eAB \leq T + tsdB$, so $eAB - tsdB \leq T$, so she could not have submitted it by $eAB - tsdB - 1 < T$.

Appendix C: Timing Parameters and Proof of Correctness for the FFO-WF Protocol

C.1 Timing Parameters for the FFO-WF Protocol

The timing parameters will be defined for a CLU Alice and a DLU Bob who share an FFO-WF channel.

timeout_deadline

In the FFO-WF protocol, the *timeout_deadline* parameter gives the number of blocks after the HTLC expiry when DLU Bob offering the HTLC goes on-chain unless both parties' Commitment transactions are updated to reflect that the HTLC has timed out. The *timeout_deadline* will be set to 0 in order to guarantee that Alice's HTLC-success transaction or Bob's conflicting HTLC-timeout transaction will be on-chain by at most L after the HTLC's expiry. Thus, this setting guarantees that the latest an HTLC can force Bob to make payment is L after the HTLC's expiry.

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, it will also be fulfilled in the current hop.

Bob's *cltv_expiry_delta* parameter is set to $2L + G$. Alice does not have a *cltv_expiry_delta* parameter, as she does not route payments.

fulfillment_deadline

The *fulfillment_deadline* parameter gives the number of blocks before the HTLC expiry when the party receiving the HTLC either goes on-chain if they have the HTLC's secret and have not updated both parties' Commitment transactions to reflect that the HTLC has been fulfilled, or updates both parties' Commitment transactions to reflect that the HTLC has timed out otherwise.

Alice's *fulfillment_deadline* parameter is set to L. Bob does not have a *fulfillment_deadline* parameter, as he does not go on-chain to fulfill an HTLC offered to him based on the expiry of the HTLC.

min_final_cltv_expiry

The *min_final_cltv_expiry* parameter gives the number of blocks from the current block height to the HTLC expiry in the last hop.

Alice's *min_final_cltv_expiry* parameter is set to $L + G$. Bob does not have a *min_final_cltv_expiry* parameter, as he does not receive payments using the FFO-WF protocol.

to_self_delay

The *to_self_delay* parameter gives a relative delay that is long enough for the party setting the parameter to detect an on-chain transaction from the other party and put a transaction on-chain in response.

Bob's *to_self_delay* parameter is set to $U + L + (R+1)$ and Alice's *to_self_delay* parameter set to $I_L + U + L + (R+1)$.

I_S and I_L

As in the WF protocol, Alice sets her I_S parameter to $B + G$ and her I_L parameter to her desired value (subject to the constraint that $I_L \geq I_S$).

max_cltv_expiry

The *max_cltv_expiry* parameter gives the maximum number of blocks in the future for an expiry for an HTLC in a signed State transaction.

The *max_cltv_expiry* parameter is set to L less than the minimum of the parties' *to_self_delay* parameters.

C.2 Correctness of the FFO-WF Protocol

As was the case for the FFO protocol, the correctness of the FFO-WF protocol will be proven assuming that the channel's Funding transaction may appear on-chain at any time and that each party can force the Funding transaction to be put on-chain eventually. It will also be assumed that the parties do not both sign a Cooperative Close transaction, as the existence of such a transaction implies that both parties agree to close the channel in the given state.

The protocol's correctness will be proven for a CLU Alice and a DLU Bob who share an FFO-WF channel.

Getting a current Commitment transaction on-chain

First, note that if either party puts an old State transaction on-chain, the channel maintenance protocol and the relative delay required before putting the corresponding Commitment transaction on-chain guarantee that the other party (or some party that knows the required per-commitment key) will revoke the old State transaction by using the required per-commitment key to spend its first output. Thus, an old Commitment transaction will never be put on-chain.

Next, we will show that if Alice follows the protocol, Bob's current State transaction will never be fixed without Alice's current State transaction being fixed in an earlier block. To see this, assume for the sake of contradiction that Bob's current State transaction is fixed in block T_1 and that Alice's current State transaction is not fixed in a block earlier than T_1. Let T_0 denote the time when Alice signed Bob's Commitment transaction corresponding to his current State transaction and note that his current State transaction has an absolute timelock of $T_0 + tsdA = T_0 + I_L + U + L + (R+1) \leq T_1$. Therefore, Alice performed channel maintenance sometime between T_0 and $T_0 + I_L$ and Bob did not update the channel state off-chain as is required (because he did not revoke his State transaction that appears in block T_1). Therefore, during this channel maintenance Alice (and thus by $T_0 + I_L$) Alice submitted her current State transaction which was fixed by $T_0 + I_L + L < T_1$, which is a contradiction.

Now, we will show that if Alice follows the protocol, Bob's current Commitment transaction will never be fixed. Assume for the sake of contradiction that Bob's current Commitment transaction is fixed and note that his corresponding State transaction (which is a parent of his Commitment transaction) must also be fixed. Therefore, Alice's current State transaction is also fixed and is in an earlier block than Bob's current State transaction (as was shown above). Let T_0 denote the block containing Alice's State transaction, let $\mathrm{T}_1 > \mathrm{T}_0$ denote the block containing Bob's current State transaction, and note that Bob's current Commitment transaction cannot be submitted until $T_1 + tsdA + tsdB$, due to its relative delay. Note that Alice's protocol for putting the channel state on-chain requires that sometime between $T_0 + tsdB$ and $T_0 + tsdB + I_L$ Alice will check the blockchain to see if she can put her current Commitment transaction on-chain. When she makes this check, there are two cases:

- **Case 1:** The Funding transaction is on-chain. In this case, she submits her current Commitment transaction which is fixed by $T_0 + tsdB + I_L + L < T_0 + tsdB + tsdA < T_1 + tsdB + tsdA$, which is before Bob's current Commitment transaction can be submitted, which is a contradiction.
- **Case 2:** The Funding transaction is not on-chain. In this case, let T_F denote the block containing the Funding transaction and note that Alice's protocol for putting the channel state on-chain requires that sometime between $T_F + tsdB$ and $T_F + tsdB + I_L$, Alice will detect that the Funding transaction is fixed and she will submit her current Commitment transaction which will be fixed by $T_F + tsdB + I_L + L < T_F + tsdA + tsdB$, which is before Bob's current Commitment transaction can be submitted, which is a contradiction.

As a result, if Alice follows the protocol, she is able to guarantee that her current Commitment transaction is eventually fixed.

Finally, if Bob follows the protocol, he will eventually be able to put his current State transaction on-chain and once both his State transaction and the Funding transaction are $tsdA + tsdB$ deep in the blockchain, he can submit his current Commitment transaction. Thus, if Bob follows the protocol, he is able to guarantee that either his current Commitment transaction, or Alice's current Commitment transaction, is eventually fixed.

Resolution of an HTLC offered to Bob

Consider the case where Bob is offered an HTLC by Alice and Bob offers a corresponding HTLC to Carol in the payment's next hop, where the next hop uses the FFO protocol (if Carol is a DLU) or the FFO-WF protocol (if Carol is a CLU). In this case, we must show that if Bob follows the protocol and pays the HTLC in the hop with Carol, he is guaranteed to receive payment for the HTLC in the hop with Alice. Assume for the sake of contradiction that Bob follows the protocol and pays the HTLC in the hop with Carol, but does not receive payment for the HTLC in the hop with Alice. Let $\mathit{cltv\_expiry\_delta}$ denote Bob's $\mathit{cltv\_expiry\_delta}$ parameter, let eAB denote the expiry of the HTLC in the hop with Alice, let eBC denote the expiry of the HTLC in the hop with Carol, and note that $eAB = eBC + \mathit{cltv\_expiry\_delta} = eBC + 2L + G$.

If the hop with Bob and Carol uses the FFO protocol, it was shown in Appendix B.2 that Bob learned the HTLC's secret by $eBC + L$. On the other hand, if the hop with Bob and Carol uses the FFO-WF protocol, it was shown in the Timeout Deadline portion of Appendix C.1 that Bob learned the HTLC's secret by $eBC + L$. Thus, in either case Bob learned the secret by $eBC + L$.

As was shown above, Bob guarantees that either Alice's current Commitment transaction or Bob's current Commitment is eventually fixed.

- **Case 1:** Alice's current Commitment transaction is eventually fixed. In this case, let T_0 denote the block in which Alice's current State transaction is fixed. There are two subcases:
  - **Subcase 1a:** $T_0 \leq eBC + L - (R+1)$. In this subcase, Bob detected Alice's current State transaction and knew the HTLC's secret by $eBC + L$, so he submitted his HTLC-success transaction by $eBC + L$. Therefore, either his HTLC-success transaction or Alice's conflicting transaction was fixed by $eBC + 2L \leq eAB$, which implies that his HTLC-success transaction was fixed before Alice could submit her conflicting transaction.

    Note that when Alice's State transaction was signed, eAB was at most $tsdB - L$ in the future (due to the $\mathit{max\_cltv\_expiry}$ parameter), so Alice could not have submitted it before $eAB - tsdB + L$, so $T_0 \geq eAB - tsdB$ (as only the top R blocks can be re-ordered). Therefore, Alice's current Commitment transaction is in block eAB or later. Therefore, Bob's HTLC-success transaction was fixed by the time Alice's Commitment transaction was fixed, and the $tsdB$ relative delay for Alice to submit her transaction that conflicts with Bob's HTLC-payment transaction guarantees that his HTLC-payment transaction wins the race and is fixed, which is a contradiction.
  - **Subcase 1b:** $T_0 > eBC + L - (R+1)$. In this subcase, Bob monitored the blockchain until he detected Alice's State transaction by $T_0 + (R+1) \geq eBC + L$, at which point he knew the HTLC's secret and he submitted his HTLC-success transaction. Therefore, either his HTLC-success transaction or Alice's conflicting transaction was fixed by $T_0 + (R+1) + L \leq T_0 + tsdB$, which implies that his HTLC-success transaction was fixed before Alice could submit her conflicting transaction.

    Note that Alice's current Commitment transaction is in block $T_0 + tsdB$ or later. Therefore, Bob's HTLC-success transaction was fixed by the time Alice's Commitment transaction was fixed, and the $tsdB$ relative delay for Alice to submit her transaction that conflicts with

Bob's HTLC-payment transaction guarantees that his HTLC-payment transaction wins the race and is fixed, which is a contradiction.

- **Case 2:** Bob's current Commitment transaction is never fixed. In this case, Bob's Commitment transaction pays the HTLC value to Bob, which is a contradiction.

Resolution of an HTLC offered to Alice

Consider the case where Alice is offered an HTLC by Bob, where Alice is the destination for the payment associated with the HTLC. In this case, we must show that if Alice follows the protocol and reveals the HTLC's secret to Bob, she is guaranteed to receive payment for the HTLC. Assume for the sake of contradiction that Alice follows the protocol and reveals the HTLC's secret to Bob, but does not receive payment for the HTLC. Let T_0 denote when Alice signs Bob's transactions for the state including the HTLC and let *min_final_cltv_expiry* denote Alice's *min_final_cltv_expiry* parameter, and note that $eAB = T_0 + min\_final\_cltv\_expiry = T_0 + L + G$.

As was shown above, Alice will eventually get her current Commitment transaction fixed.

Note that Alice was unable to update the channel state to reflect the payment of the HTLC to her (given the assumptions above), so she submitted her current State and HTLC-success transactions at $T_0 + G$. Therefore, either Alice's HTLC-success transaction or Bob's conflicting HTLC-timeout transaction is fixed by $T_0 + L + G \leq eAB$, which implies that her HTLC-success transaction was fixed before Bob could submit his conflicting HTLC-timeout transaction.

Therefore, both Alice's current Commitment transaction and her associated HTLC-success transactions are eventually fixed, which implies that Bob's HTLC-refund transaction can never be fixed, so Alice will eventually receive the payment for the HTLC, which is a contradiction.