# Resizing Lightning Channels Off-Chain With Hierarchical Channels

John Law

September 6, 2023

Version 1.3

**Abstract**

Dynamic management of Lightning channel capacity is required to support efficient Lightning payments. On-chain resizing introduces delays, adds costs and limits scalability. Furthermore, fast and cheap resizing of channels may be required in order to provide watchtower-free Lightning payments for casual users. While channel factories have been proposed for resizing channels off-chain, existing proposals only exchange capacity within a single factory. As a result, their utility is greatly limited.

This paper introduces hierarchical Lightning channels that allow users to resize Lightning channels flexibly and off-chain, thus doing for channel capacity what Lightning does for bitcoin. In addition, hierarchical channels allow a casual user to share channel capacity with a pair of dedicated users, both of whom can use all of their channel capacity to route Lightning payments even when the casual user is not active. As a result, casual users can have watchtower-freedom without stranding any Lightning capacity. No change to the underlying Bitcoin protocol is required.

## 1 Introduction

The Lightning Network (LN) is a network of 2-user channels that allow users to send and receive bitcoin in a trust-free manner without putting any transactions on the Bitcoin blockchain in the usual case **[BOLT]**. The ability to keep most transactions off-chain greatly increases Bitcoin's scalability, reduces fees, and makes payments nearly instantaneous. However, the LN alone does not fully solve Bitcoin's scalability problems, as on-chain transactions are typically required in order to open, close and resize Lightning channels.

Dedicated users route Lightning payments to and from casual users. In particular, a casual user and a dedicated user can create a channel (called a *payment channel*) for payments to and from the casual user, while a pair of dedicated users can create a channel (called a *routing channel*) for routing payments for others. Both payment channels and routing channels must be resized in order to move

channel capacity to where it can be utilized most efficiently. In fact, it has been stated that the "main (probably only) job of" a dedicated user "is to efficiently allocate their liquidity" **[Tei22]**. The use of on-chain transactions to resize channels limits the LN's scalability and increases its fees. In addition, resizing channels on-chain can introduce substantial delays. In fact, delays of multiple months are possible in the case of a casual user that operates in a watchtower-free manner **[Law22a]**, thus stranding capital and potentially preventing such users from operating in a watchtower-free manner in the current environment **[Tei22]**.

Existing proposals for resizing channels off-chain consist of creating a channel factory **[BDW18]** or CoinPool **[NR]** and exchanging capacity between channels within the same factory or pool. However, only a very small fraction of the LN's channels can be expected to be in a single factory or pool **[Ria22]**, so the ability to match channels that have insufficient capacity with channels that have excess capacity is highly constrained.

This paper uses hierarchical Lightning channels to solve two problems.

First, as long as routing channels are created within hierarchical channels, it is possible to resize them flexibly, nearly instantly and off-chain. Thus, hierarchical channels do for routing channel capacity what Lightning does for bitcoin. In fact, this is more than just an analogy, as the routing channel capacity is actually transferred over the LN.

Second, hierarchical channels can be created by a casual user and a pair of dedicated users such that the casual user can send and receive bitcoin in a watchtower-free manner, while the dedicated users can use all of their channel capacity to route payments even while the casual user is inactive. As a result, casual users can operate in a watchtower-free manner without stranding any capital.

Neither result requires any change to the underlying Bitcoin protocol.

The remainder of this paper is organized as follows. Section 2 defines hierarchical channels and presents necessary terminology. Section 3 demonstrates how hierarchical channels can resize routing channels off-chain and Section 4 shows how hierarchical channels can be used to support watchtower-free casual users without stranding capital. Protocols for implementing hierarchical channels are given in Section 5. Sections 6, 7 and 8 present a discussion of hierarchical channels, related work and conclusions.

# 2 Hierarchical Channels

A p*arty* is a group of one or more users. A *hierarchical channel* is a 2-party channel that has two main outputs, one per party, plus zero or more Hash Time-Locked Contract (HTLC) outputs. Each output from a hierarchical channel that pays to a multi-user party funds another (potentially hierarchical) channel. As a result, each output in a hierarchical channel (including an HTLC output once it has been resolved) can be viewed as the root of an off-chain tree of outputs where the leaves are owned by single users.

In order to update a hierarchical channel, funds are offered by one party to the other party in an HTLC. One user within the party offering the HTLC is designated as the *payer* and one user within the party offered the HTLC is designated as the *payee*. All of the funds for the HTLC are provided by the payer, and if the HTLC succeeds, the bulk of the funds go to the payee (but users within the offered party other than the payee can also get routing fees). Before the channel state is updated to include a new HTLC output, all of the users in the channel sign new transactions that spend the new channel state's main outputs, its existing HTLC outputs, and the new HTLC output. The users then sign transactions that implement the new channel state (including the new HTLC output) and revoke the previous channel state. Once the HTLC is resolved, the channel state is updated to include the HTLC's funds in the offered party's main output (if the HTLC succeeded) or in the offering party's main output (if the HTLC failed).

Because the two parties within a hierarchical channel can use an HTLC to exchange bitcoin, they can link their HTLC to HTLCs in other (potentially hierarchical) channels, thus making payments over the LN. In particular, each party in a (potentially hierarchical) channel appears as a node in the LN channel graph and each (potentially hierarchical) channel appears as a pair of unidirectional edges linking the channel's two parties. As in the current LN, a payment consists of a path where the party that is offered an HTLC in one hop offers an HTLC in the next hop (and the user that is the payee in one hop is the payer in the next hop)[1].

# 3 Resizing Routing Channels Off-Chain

## 3.1 Reducing The Size Of A Routing Channel

Dedicated users who provide routing services to others can use hierarchical channels to resize their routing channels off-chain. For example, consider dedicated users A, B, C and D who create a hierarchical channel with parties AB and CD. This hierarchical channel's main outputs fund the (non-hierarchical) routing channels owned by AB and by CD. Also, assume that party CD has a separate hierarchical channel with (single-user) party E.

If AB wants to reduce the capacity of their (non-hierarchical) channel, they can create a payment through the LN from party AB (with user A being the payer) to party (and user) A consisting of the path AB -> CD -> E -> F -> G -> A.

An example of such a payment is shown in Figure 1 below.

1 Each party implementing the payment can be a single-user party or a multi-user party, and a single payment can utilize an arbitrary mix of single- and multi-user parties.

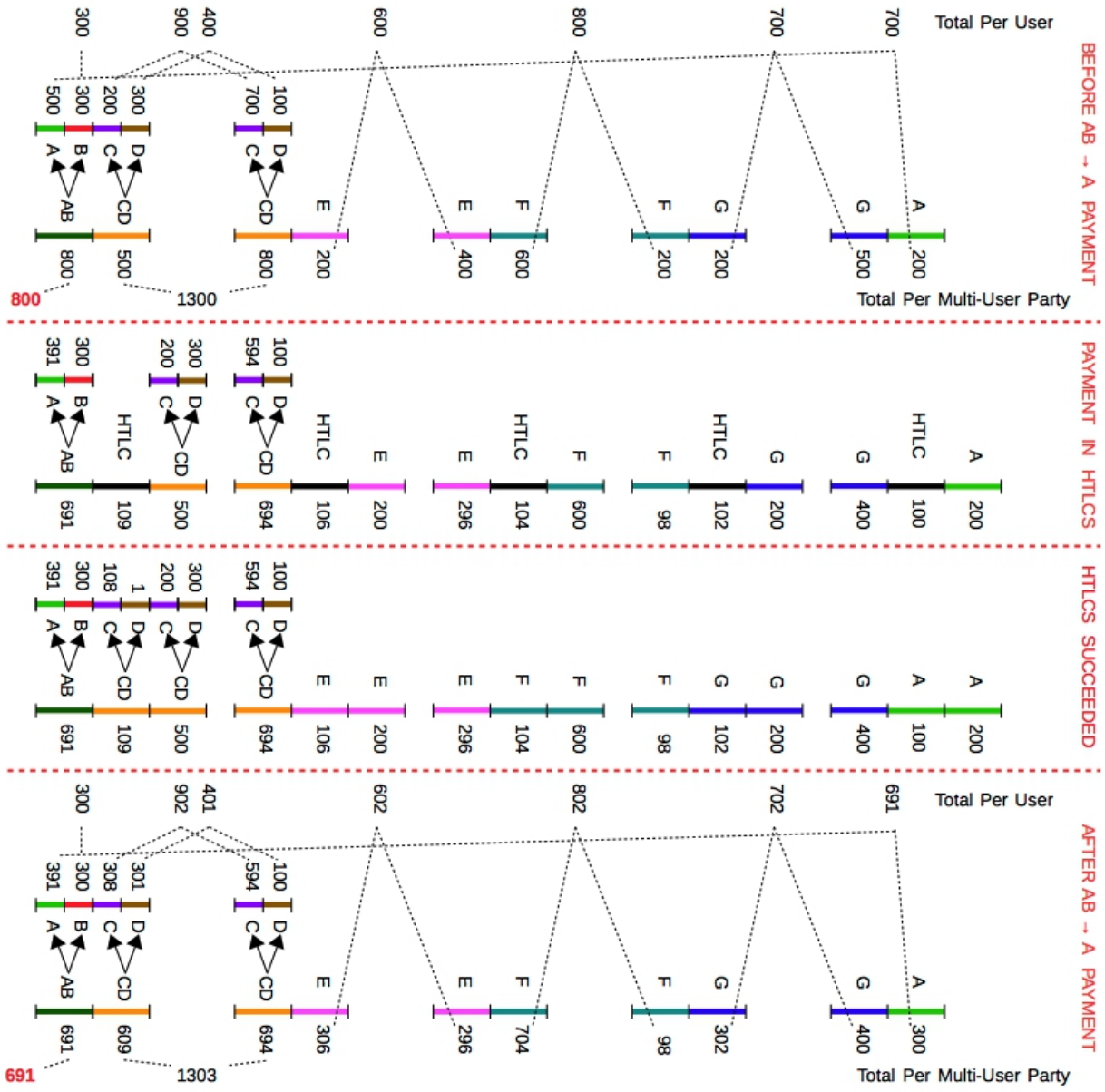


**Figure 1.** The size of the routing channel owned by AB is reduced by making a Lightning payment of 100 units (plus fees) from party AB to party A via parties CD, E, F and G. Each colored bar represents an output from a Lightning channel. Funds are held in HTLCs in the middle two panels, while the bottom panel shows the final result of the payment. Routing users that fund HTLCs (payers) receive a fee of 2 units, while user D receives a fee of 1 unit for providing the necessary signatures. Users A and B do not receive fees, but do agree to (and thus benefit from) owning a smaller channel.

Because this payment is from party AB within the hierarchical channel owned by parties AB and CD, this payment reduces the value of the main output that pays to AB from this hierarchical channel. Furthermore, because the (non-hierarchical) routing channel owned by A and B is funded by the main

output that pays to AB from that hierarchical channel, this payment reduces the size of the channel owned by A and B. Finally, because this payment is paid by A (as the payer within the offering party AB at the payment's first hop) and pays to A (as the payment's destination), user A neither gains nor loses funds (except for paying routing fees to users C, D, E, F and G).

Furthermore, as in any LN payment, each routing node that gains funds in one hop loses an identical (minus fees) amount of funds in the next hop, so their overall balance is unchanged (except for a slight increase due to fees). There is nothing new about this for routing nodes E, F and G, as they act as they do in the current LN. However, it is worth looking in more detail at routing node CD, as the existence of multi-user nodes in the LN is new with hierarchical channels.

Like any routing node, CD gains funds in the hierarchical channel with AB and loses an identical (minus fees) amount of funds in the hierarchical channel with E. Note that C and D also appear as nodes within the LN channel graph, the channel between C and D that is funded by the hierarchical channel with AB appears as a pair of unidirectional edges between C and D, and the channel between C and D that is funded by the hierarchical channel with E appears as a pair of unidirectional edges between C and D. Thus, the sum of the capacities of the two channels between C and D is unchanged, except for a slight increase due to fees.

## 3.2 Logical vs. Physical Channels

In the above example, because there are two separate channels between C and D (one funded by the hierarchical channel with AB and the other funded by the hierarchical channel with E), it is natural to express these channels as two separate pairs of edges between C and D. However, a better viewpoint is obtained by logically merging the two *physical channels* between C and D into a single *logical channel* containing the sum of the capacities of the two physical channels. In fact, a large payment can be routed between C and D by using parallel HTLCs in the two physical channels connecting them. Also, when logical (rather than physical) channels are advertised to LN peers, the resulting channel capacities are much more stable, as they do not change (other than increasing due to fees) when the parties owning the channel are used to route a payment.

The effects of this payment from AB to A on a portion of the LN channel graph are shown in Figure 2 below. Each party appears as a box and each channel appears as a pair of directed edges between a pair of parties. The channel's capacity is shown in magenta between the pair of directed edges, and each party's balance appears in red either above or below the pair of edges. Viewpoints based on both physical channels and logical channels are shown. The funds owned by each party are shown within the party's box.

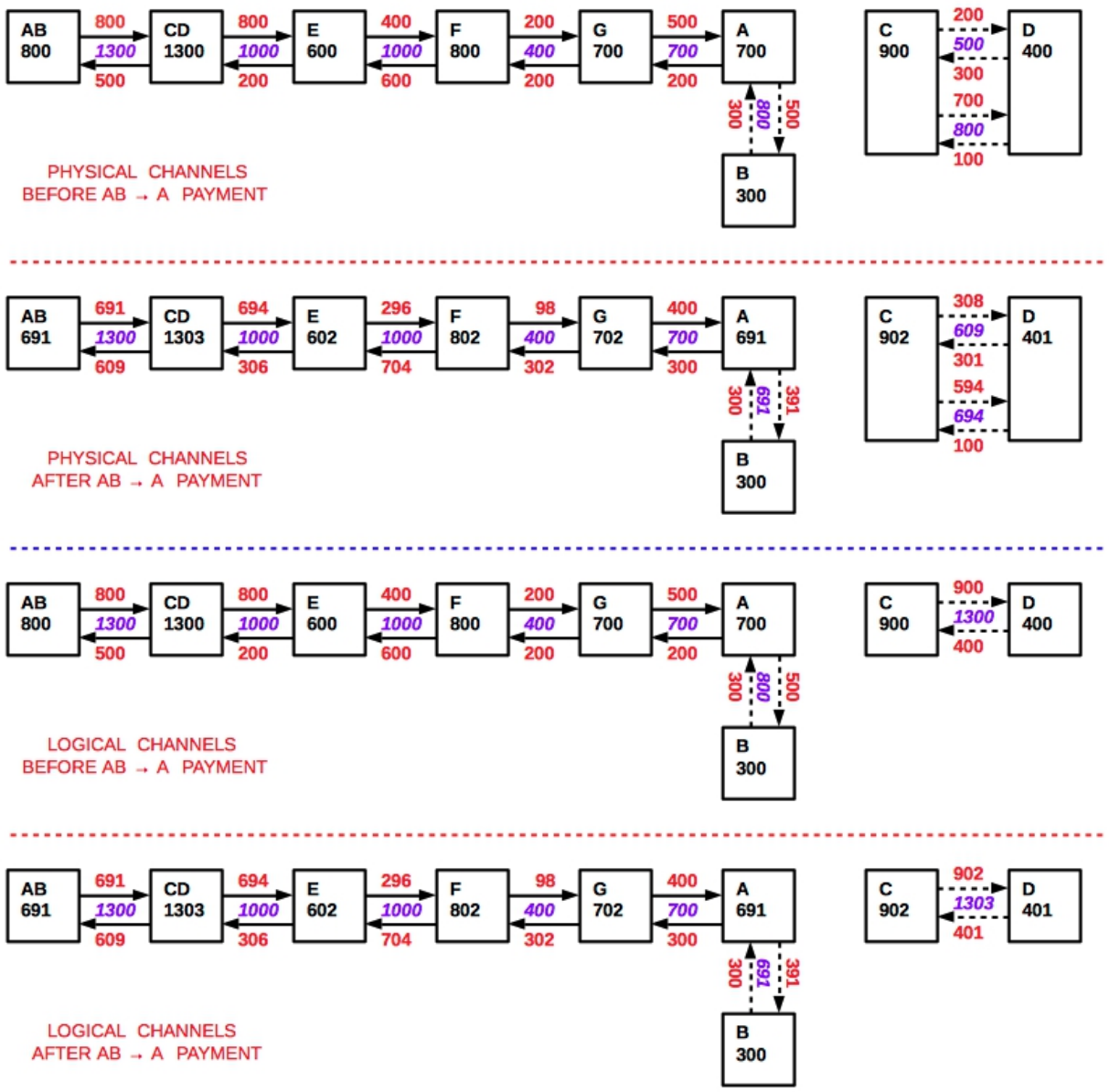


**Figure 2.** This figure shows how a portion of the Lightning channel graph is updated by the payment in Figure 1. Each node represents a party in a (possibly hierarchical) channel. Each channel is represented by a pair of arcs between a pair of nodes, with channels funded on-chain having solid arcs and channels funded off-chain having dashed arcs. Channel capacities are shown in magenta and channel balances are shown in red. The total of all balances owned by a party within this portion of the channel graph are given in black. The upper half of the figure shows physical channels, while the lower half shows logical channels. Within each half, the upper panel shows channels and nodes prior to the payment and the lower panel shows them after the payment.

While the above figures show a successful payment that reduces the size of a routing channel, it is possible (though unlikely) that the payment will fail after the HTLCs are put in place. Such a payment failure is depicted in Figure 3 below.

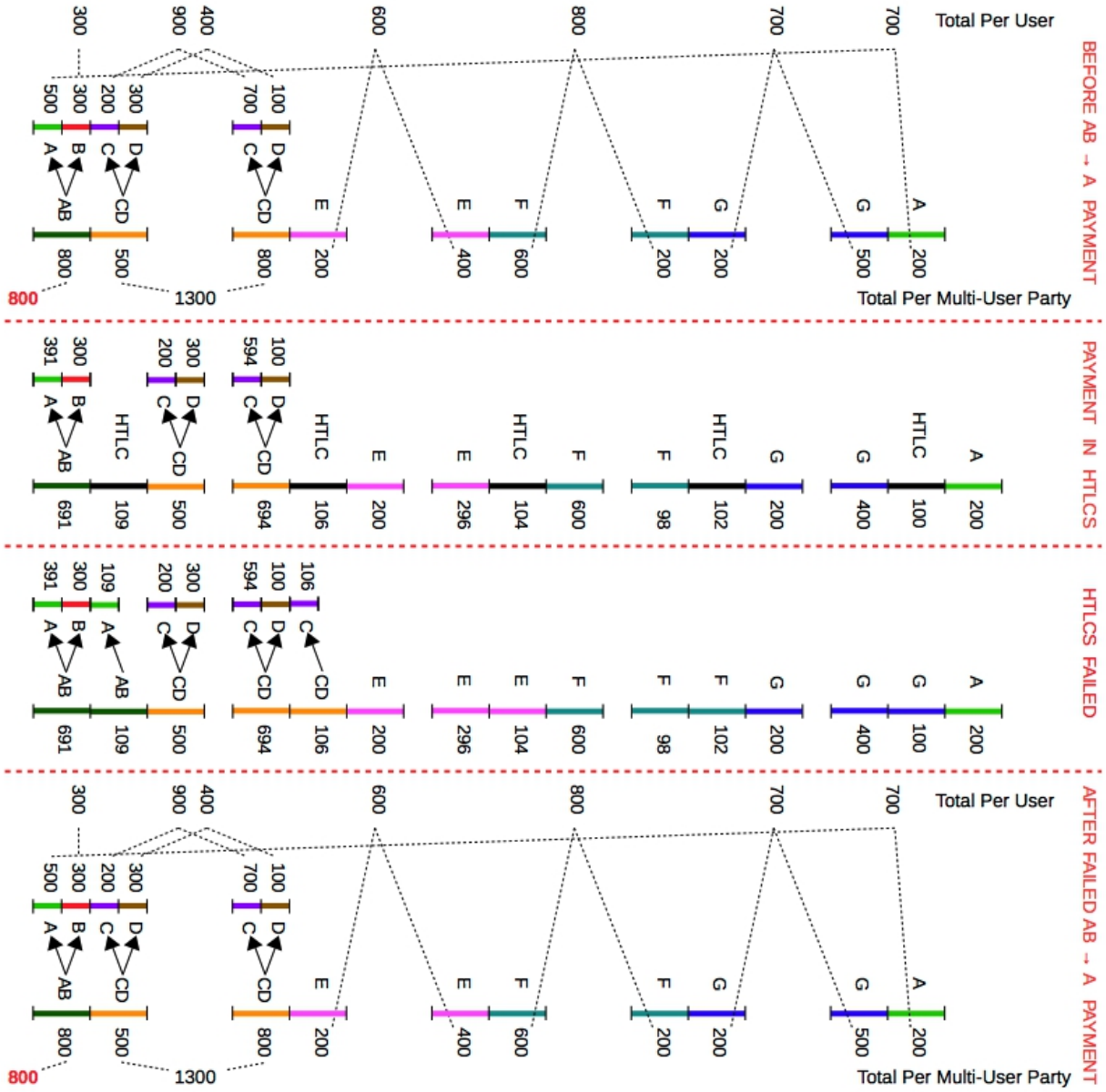


**Figure 3.** The same payment as in Figure 1 showing the case where the payment fails due to a timeout of the HTLC offered by G to A. All parties and users end up with the same balances as they had prior to the failed payment.

Figure 1 depicts a payment that decreases the capacity of logical channel AB (which appears within a 4-user channel) by routing the payment through 3-user and 2-user channels. However, the payment could also be routed through one or more additional 4-user channels, as is shown in Figure 4 below.

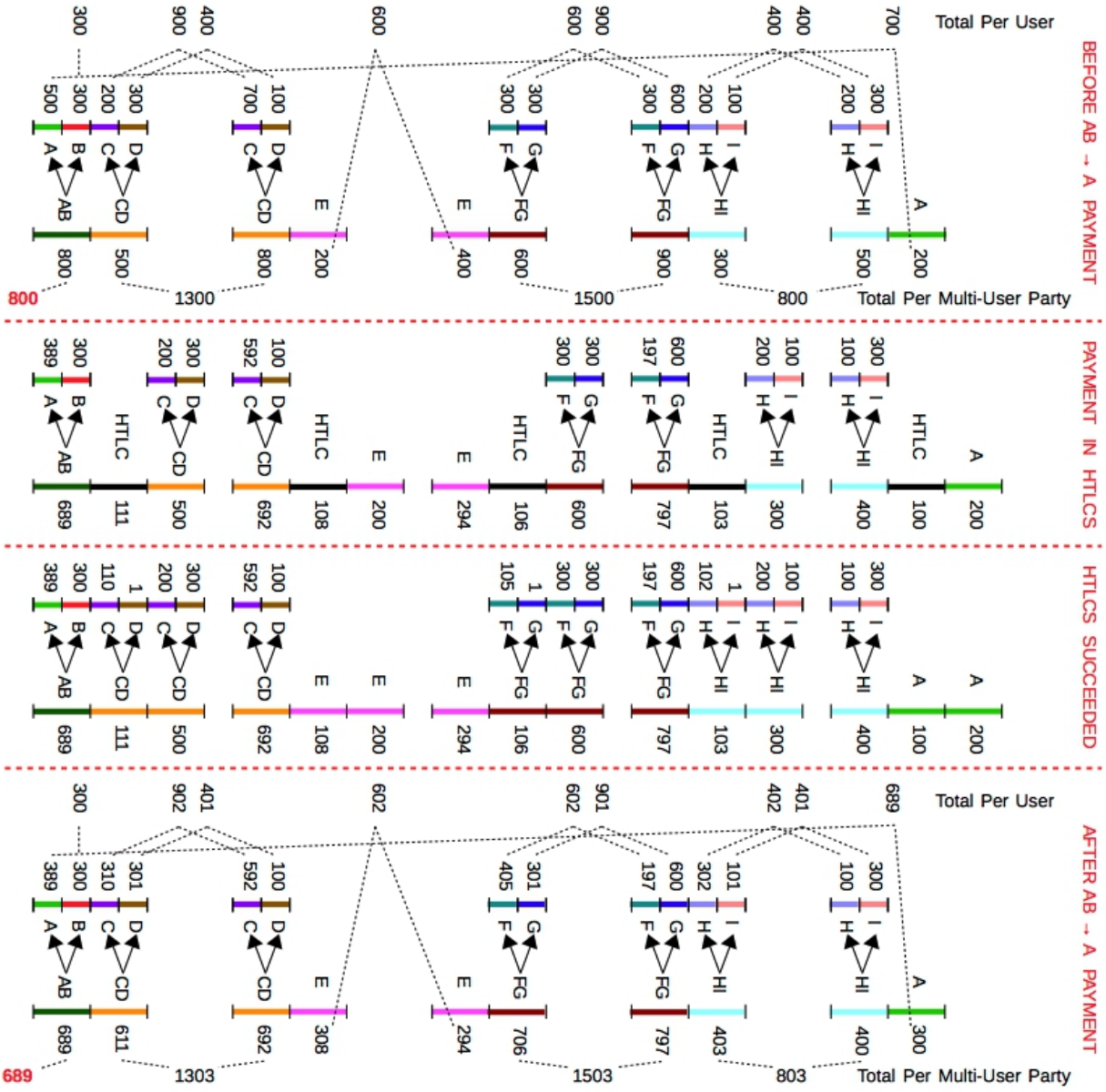


**Figure 4.** This figure shows another example of how the size of logical channel AB can be reduced by making a Lightning payment of 100 units (plus fees) from party AB to party A. In this example, the payment is routed through 3-user and 4-user hierarchical channels.

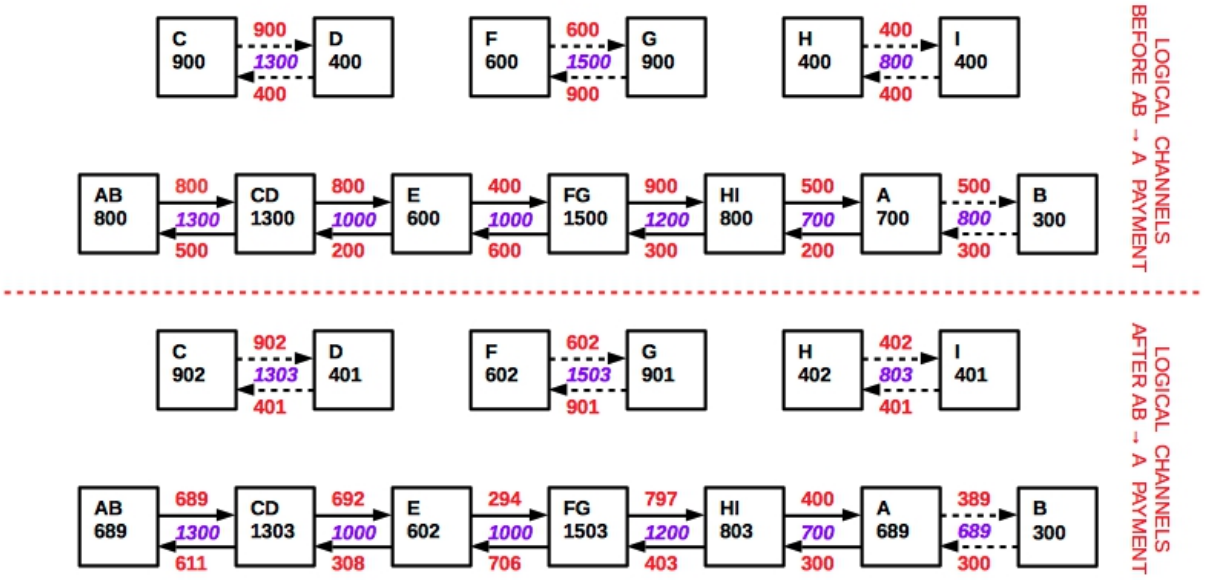


**Figure 5.** This figure shows how a portion of the Lightning channel graph (using logical channels) is updated by the payment in Figure 4.

## 3.3 Increasing The Size Of A Routing Channel

Of course, the example from Figure 1 could be reversed (by making a payment from A to AB) in order to increase the capacity of the logical channel between A and B off-chain. An example of such a payment is shown in Figures 6 and 7 below.

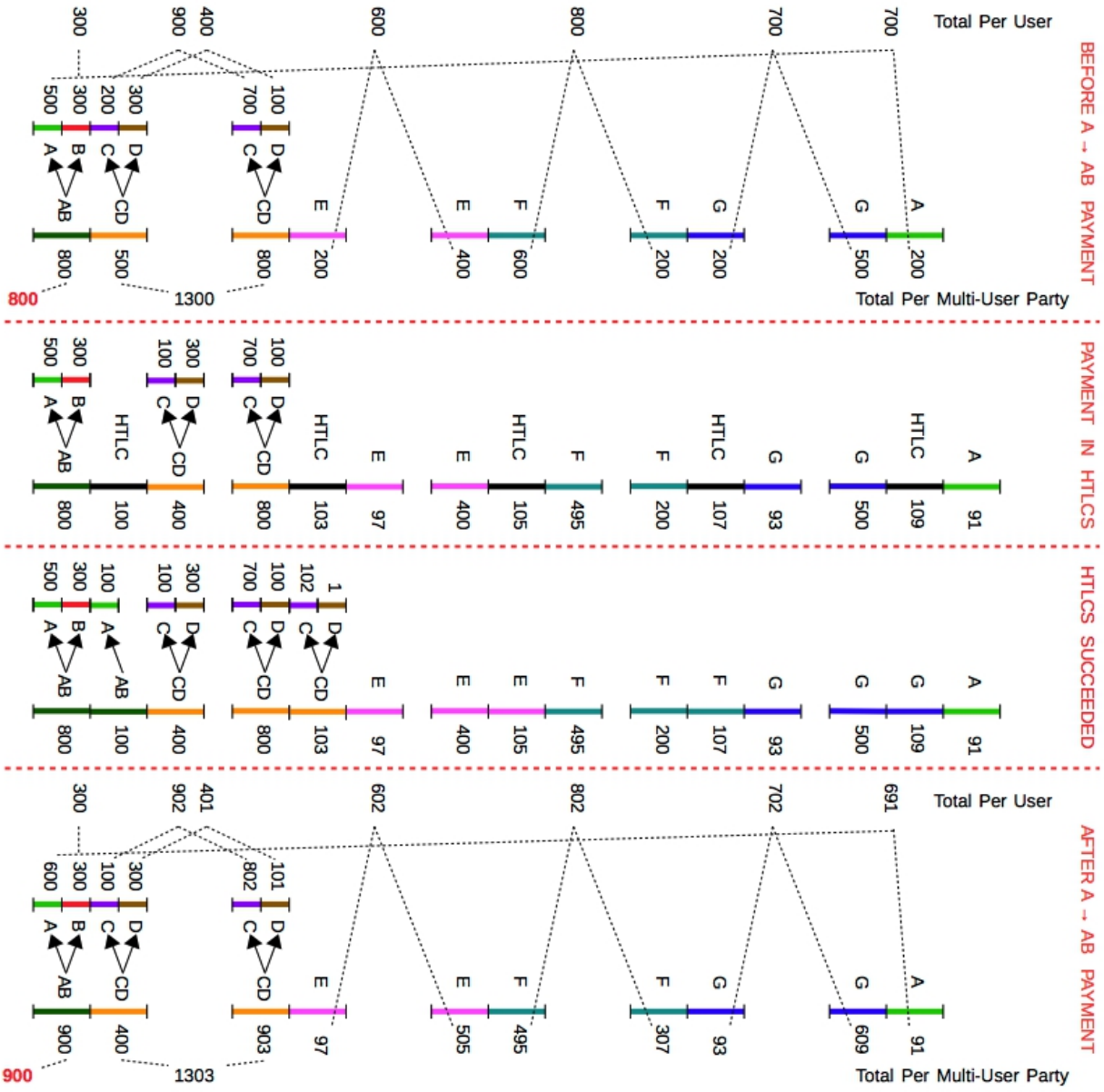


**Figure 6.** The size of routing channel AB is increased by making a Lightning payment of 100 units (plus fees) from party A to party AB via parties G, F, E and CD.

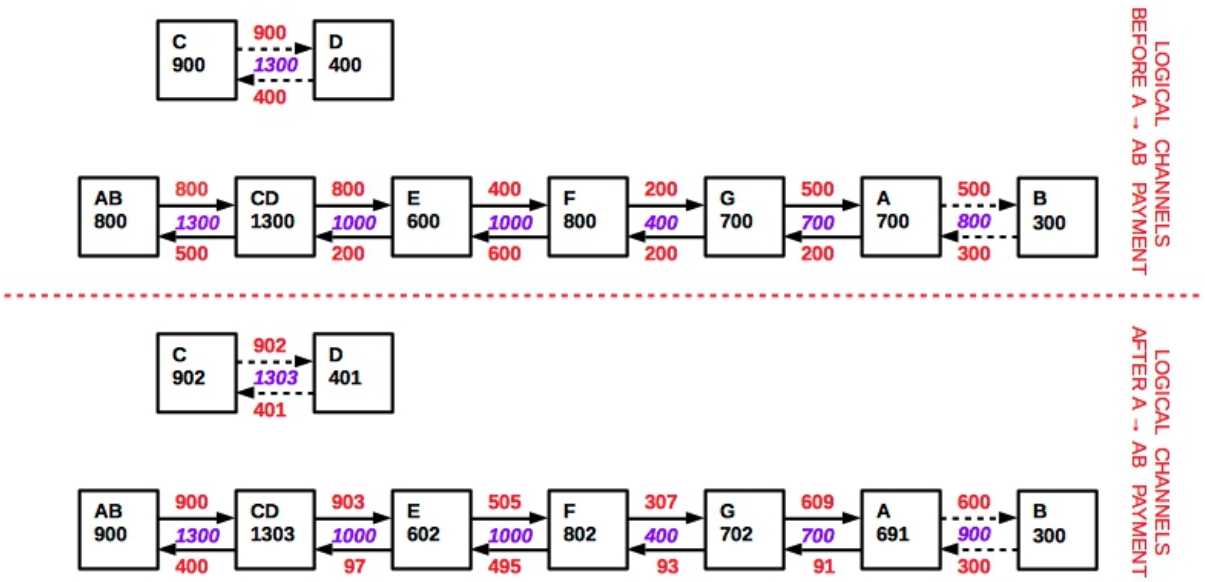


**Figure 7.** This figure shows how a portion of the Lightning channel graph (using logical channels) is updated by the payment in Figure 6.

## 3.4 General Rules for Resizing Channels Off-Chain

In general, a payment on the LN removes funds from the source of the payment, add funds to the destination of the payment, and leaves the routing nodes traversed by the payment unchanged (except for fees). With hierarchical channels, the source, the destination, and/or the routing nodes can be multi-user parties.

In order to reduce the size of a logical channel L off-chain:

1. select a physical channel P within L, where the P is funded by an off-chain UTXO that is the output of a hierarchical channel H,
2. select a user U within the party owning P,
3. use the LN to send a payment from the party owning P to U, where the first hop of the payment is within H and U is the payer within that hop.

In order to increase the size of a logical channel L off-chain, perform the same three steps, except in Step 3 send the payment from U to the party owning P, where the last hop of the payment is within H and U is the payee within that hop.

When reducing the size of a logical channel, the amount of the payment equals the amount by which the size is reduced. When increasing the size of a logical channel, the amount of the payment equals the amount by which the size is increased, plus fees.

Note that the above rules require that either the payment's source or its destination (but not both) is a 1-user party. As a result, at least one of the nodes involved in the payment must have a 1-user party. Furthermore, assume there are only 4-user channels (owned by a pair of 2-user parties), 3-user channels (owned by a 2-user party and a 1-user party) and 2-user channels (owned by a pair of 1-user parties). In this case, a 2-user channel can only be resized off-chain if it is funded by an off-chain UTXO that is the output of a 4-user channel or 3-user channel, and the payment that resizes it must involve a 3-user channel (in order to transition between multi-user parties and 1-user parties).

# 4 Supporting Casual Users Without Stranding Capacity

Hierarchical channels can also be used to support watchtower-free casual users without stranding any channel capacity. Consider a watchtower-free casual user A who creates a hierarchical channel with dedicated users B and C, where A is one party and BC is the other party in the channel. Assume A wants to make a payment to G, and that party BC has a separate hierarchical channel with user D, D has a channel with E, E has a channel with F, and F has a channel with G. In this case, A can create a payment through the LN consisting of the path A -> BC -> D -> E -> F -> G.

An example of such a payment is shown in Figures 8 and 9 below.

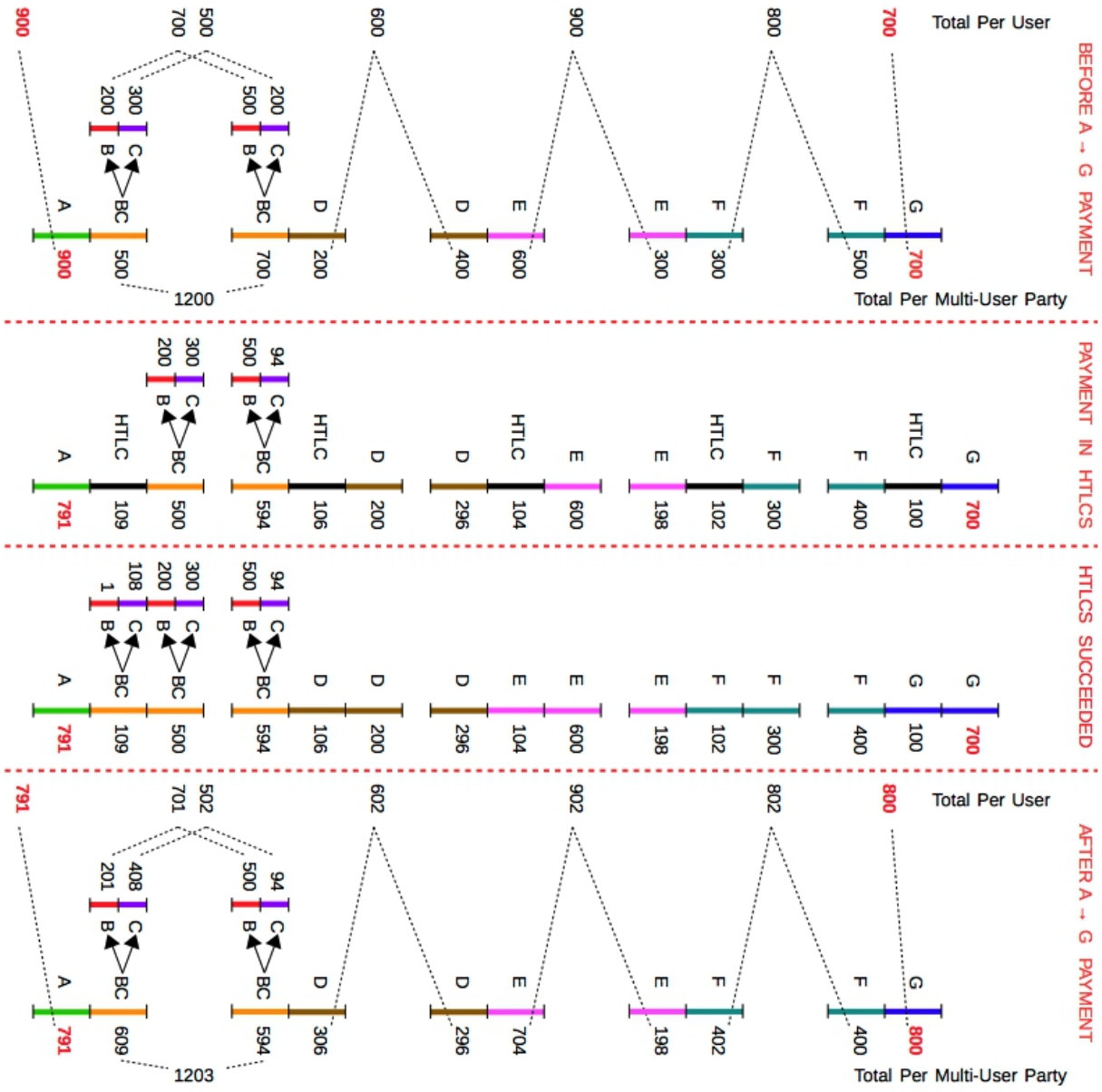


**Figure 8.** The payment of 100 units (plus fees) from casual user A to user G routing through parties BC, D, E and F.

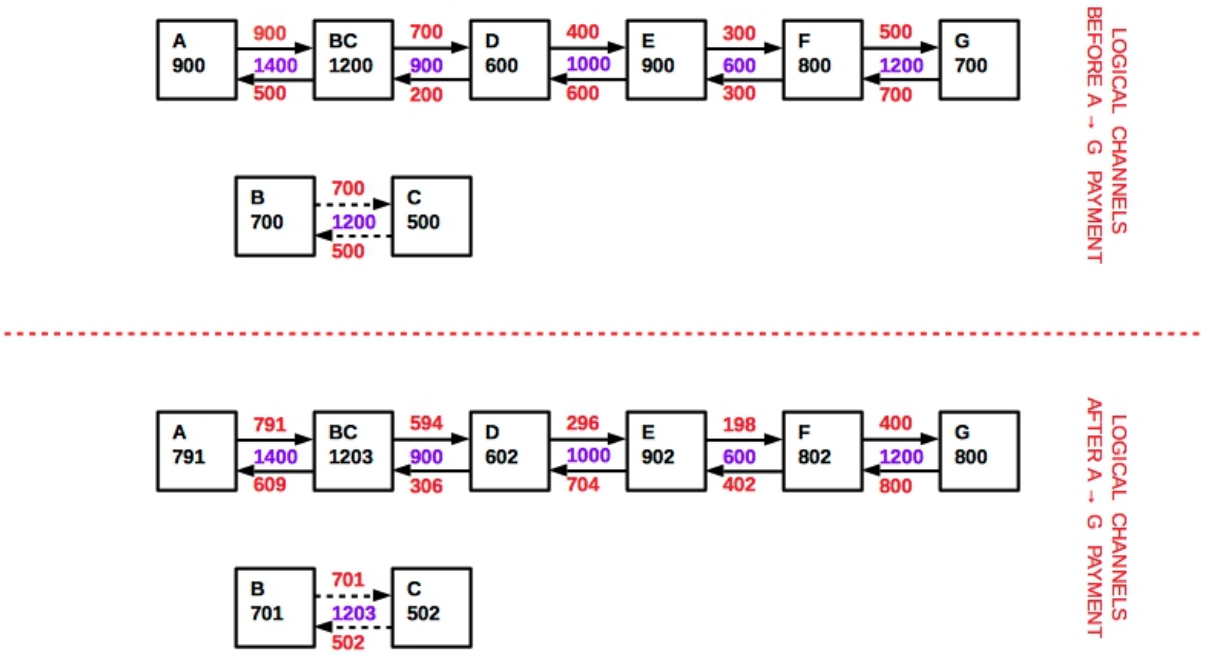


**Figure 9.** Portion of the logical channel graph showing the payment in Figure 8 from casual user A to user G via parties BC, D, E and F.

Of course, this example could be reversed in order to make a payment from G to casual user A, as is shown in Figures 10 and 11 below.

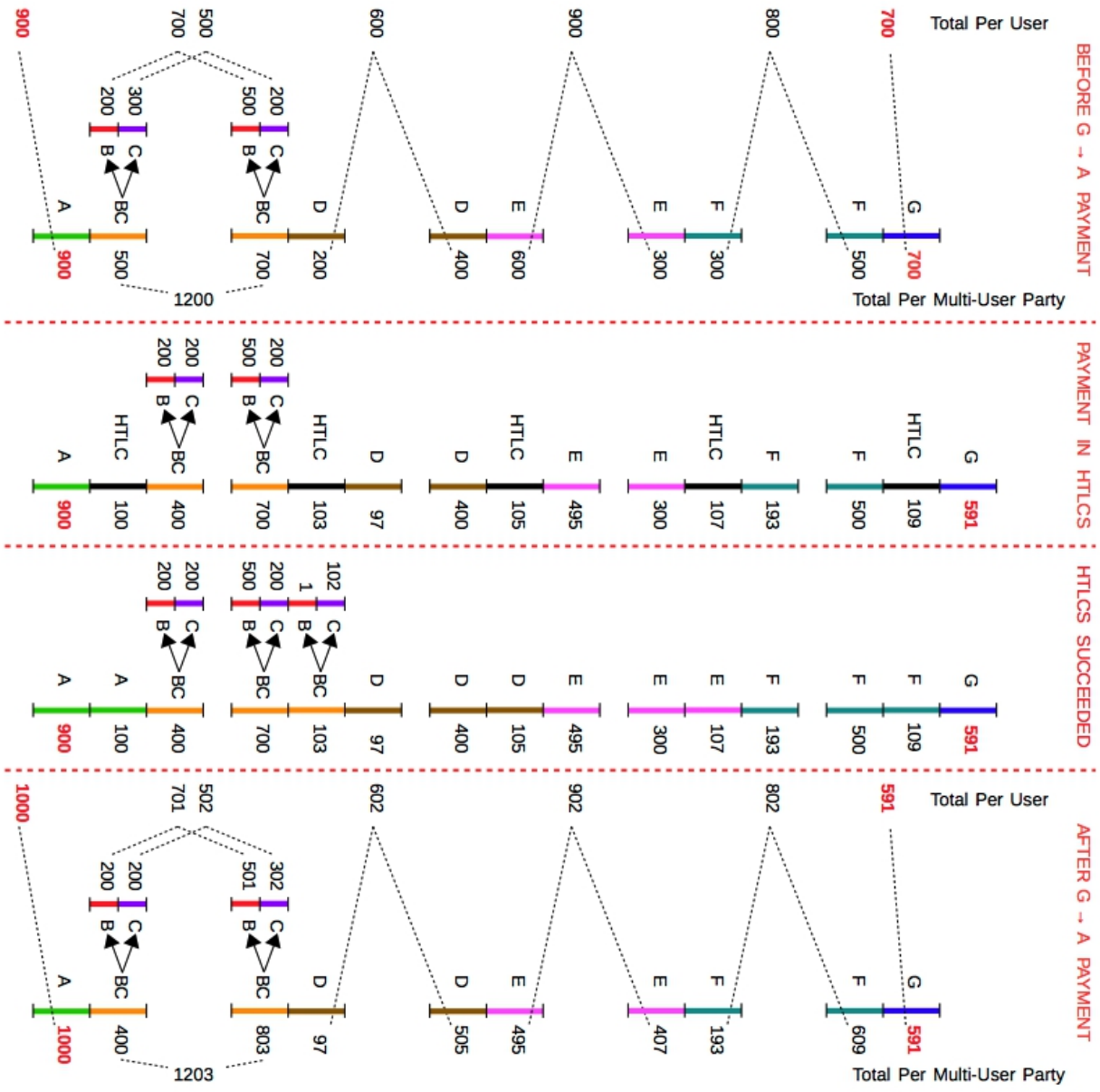


**Figure 10.** The payment of 100 units (plus fees) to casual A from user G routing through parties F, E, D and BC.

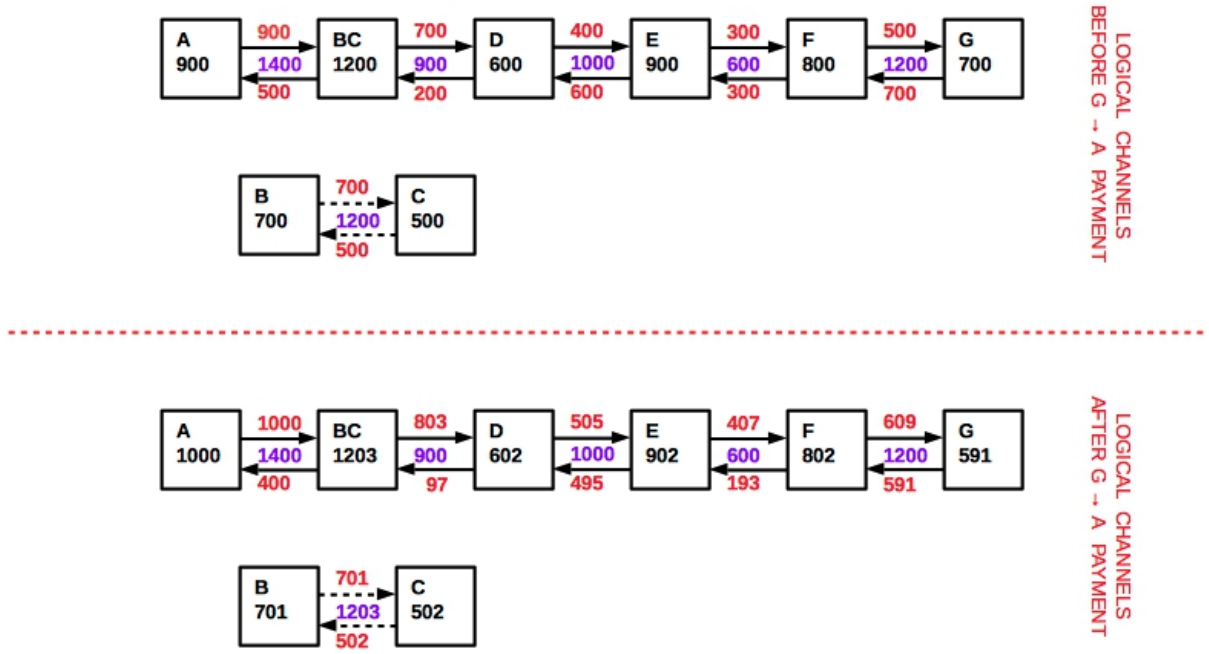


**Figure 11.** Portion of the logical channel graph showing the payment in Figure 10 to casual A from user G via parties F, E, D and BC.

The key observation is that whenever A is neither sending nor receiving a payment, dedicated users B and C can utilize all of the capacity in the channel between them to route LN payments. As a result, no channel capacity is stranded while casual user A is offline.

Also, note that as long as B and C announce their two physical channels (namely the one funded by the hierarchical channel shared with A and the one funded by the hierarchical channel shared with D) as a single logical channel, the capacity of the logical channel is unchanged (other than in increase due to fees) when B and C route payments to and from A.

# 5 Protocols For Hierarchical Channels

## 5.1 Overview

Several issues have to be addressed in order to create a protocol for hierarchical channels.

First, each main output (or resolved HTLC output) from a hierarchical channel must be able to pay to a tree of transactions that distribute the output's funds to individual users at the leaves. This can be achieved by requiring the users in the hierarchical channel to exchange signatures for the tree of transactions that spend each output of a given Commitment transaction prior to signing that Commitment transaction.

Second, a Lightning channel is currently announced with a *channel_announcement* message that references the on-chain UTXO funding the channel and the channel's capacity is static **[BOLT]**. These *channel_announcement* messages will not work for hierarchical channels that are funded by off-chain UTXOs and have dynamic capacities. Russell's proposal **[Rus22]** for *channel_update_v2* messages is well-suited to hierarchical channels as it merges the concepts of channel announcements and channel updates and it includes the channel's capacity, thus supporting dynamic capacities. In addition, it supports channels that are funded off-chain by allowing on-chain UTXOs to be cited when announcing channels with capacities that are at most some fixed multiple of the on-chain UTXOs' value.

Third, hierarchical channels need to support more than two users per channel. The current Lightning channel protocol **[BOLT]** only works for 2-user channels, as it penalizes a user that puts an old Commitment transaction on-chain by allowing the other user to obtain all of the channel's funds. Such an approach does not work if there are more than two users in the channel, where at least two are dishonest. In such a case, a dishonest user could put an old Commitment transaction on-chain and another dishonest user could "punish" that user by claiming all of the channel funds, including those from the honest users. Even if the dishonest users cannot guarantee that they will take the channel's funds, the expected value obtained by dishonest behavior could exceed the expected value from honest behavior.

In contrast, channel factories **[BDW18]** are designed to allow more than two users to update factory states off-chain, where each factory state consists of a division of the factory's funds among the factory's users. Therefore, if we define the hierarchical channel's state as consisting of its two main outputs, plus any HTLC outputs, a channel factory protocol can be used to update the channel's state off-chain. In particular, the Invalidation Tree protocol **[DW15]**, the Tunable-Penalty Factory protocol **[Law23]** or the Single-Commitment Factory protocol **[Law23]** can be used. Of these, the Tunable-Penalty Factory protocol appears most attractive, as it requires only O(1) time and O(1) on-chain bytes for a unilateral close.

Finally, in addition to maintaining the channel's current set of outputs, each HTLC output must be resolved according to the terms of its associated HTLC. A simple approach, based on the current Lightning protocol, is to allow the HTLC output to be spent by either:

- an HTLC-success transaction put on-chain by the payee and providing the HTLC's required preimage, or
- an HTLC-timeout transaction put on-chain by any user within the offering party after the HTLC's expiry.

This approach works, but it has two serious performance problems:

1. resolving an HTLC on-chain requires closing the hierarchical channel, and
2. the latency required for putting the HTLC output on-chain can significantly delay the HTLC's expiry.

Fortunately, both of these performance challenges are solved by using separate control transactions to resolve the HTCLs, as is done in the FFO and FFO-WF channel protocols **[Law22b]**. Specifically, the FFO protocol can be extended to resolve HTLCs in hierarchical channels owned by more than two dedicated users, and the FFO-WF protocol can be extended to resolve HTLCs in hierarchical channels with more than two users, exactly one of whom is a casual user. The details of these constructions are given in the remainder of this section.

## 5.2 Notation

In the following figures:

- **{A, B, C, D, E}** denote that the given user's signature is required,
- **pckey{A|B|C|D|E}i** denotes a signature using a per-commitment key for revoking the given party's state *i* transaction,
- ***e*{A, B, C, D, E}** denotes the expiry for the current payment in the channel shared by the given users,
- ***tsd*{A, B, C, D, E}** denotes the maximum of the *to_self_delay* channel parameters set by the given users, and
- **Preimage(X)** denotes the current payment's secret (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,
- **Com** for a Commitment transaction,
- **In** for an Individual transaction,
- **St** for a State transaction,
- **H-k** for an HTLC-kickoff transaction,
- **H-s** for an HTLC-success transaction,
- **H-t** for an HTLC-timeout transaction,
- **H-p** for an HTLC-payment transaction, and
- **H-r** for an HTLC-refund transaction.

All of the above transactions have the same functionality as in the Tunable-Penalty Factory protocol **[Law23]**, the FFO protocol **[Law22b]**, and the FFO-WF protocol **[Law22b]**. While the Tunable-

Penalty Factory, FFO, and FFO-WF protocols are reviewed briefly below, knowledge of those protocols is required for a full understanding of the hierarchical channel protocols presented here.

Subscripts denote which user(s) can put the transaction on-chain (if not all users can do so) and which channel state the transaction is associated with. Transactions that can only be put on-chain by certain users are denoted with color, and those that can be put on-chain by any user are black.

Bold lines carry channel funds, thin solid lines have value equal to, or slightly larger than[2], a tunable penalty amount or a fee 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 user 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. 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.

## 5.3 4-User Channels

First, consider a hierarchical routing channel owned by four dedicated users, two per party. The channel's state, which consists of the funds that pay to each of the channel's main outputs (and its zero or more HTLC outputs) is maintained using the Tunable-Penalty Factory protocol **[Law23]**. Each user has an Individual transaction with an output that can be spent by that user by putting a State transaction on-chain. The State transaction is a control transaction that determines which Commitment transaction the user can put on-chain, as each Commitment transaction spends the output of a unique State transaction. Revoked Commitment transactions are prevented from being put on-chain by allowing a window of time during which other users can spend the output of the State transaction that is needed by the Commitment transaction. More precisely, the first output of each State transaction can be spent by any user that knows the per-commitment key associated with it. Therefore, a user can revoke their Commitment transaction for a given state $i$ by sharing the per-commitment key for state $i$ with the other users in the channel.

A 4-user channel that uses the Tunable-Penalty Factory protocol is shown in Figures 12 and 13 below.

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

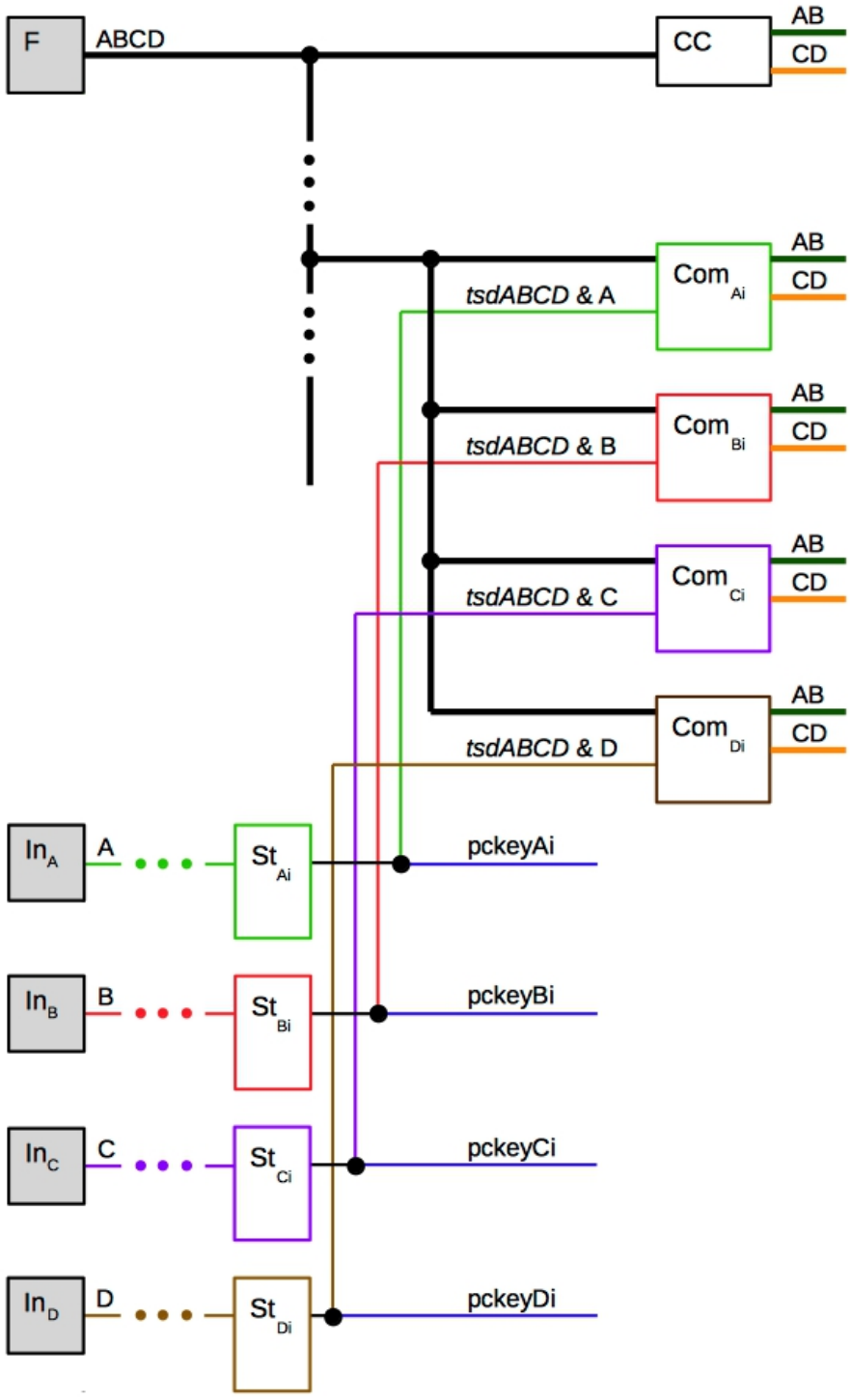


**Figure 12.** 4-user hierarchical channel with outputs to AB and to CD. The output to AB funds a Lightning channel owned by dedicated users A and B, while the output to CD funds a Lightning channel owned by dedicated users C and D. Any of A, B, C or D can put their State and Commitment transactions on-chain to unilaterally close the channel in its current state.

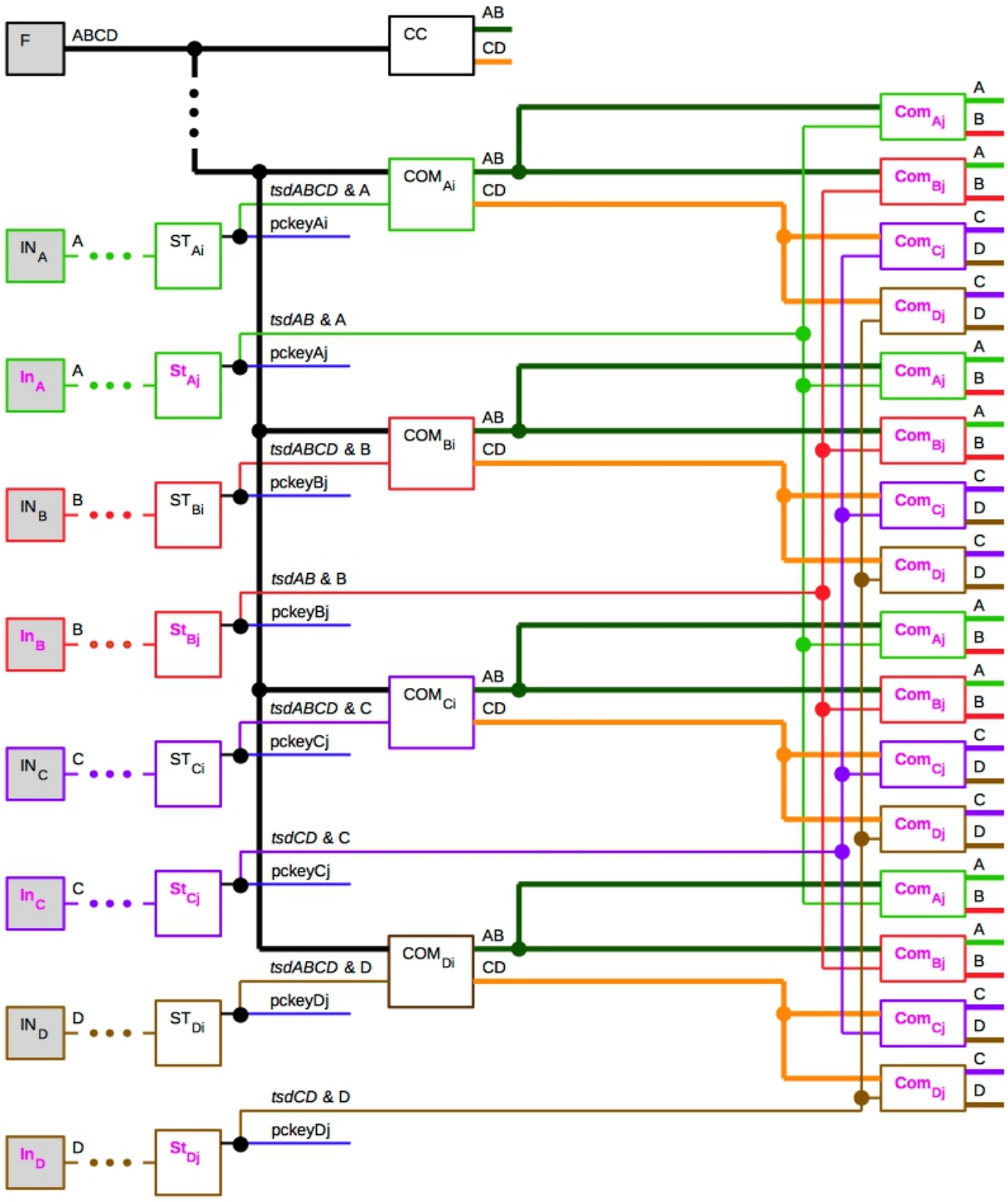


**Figure 13.** 4-user hierarchical channel from Figure 12 showing the control and value transactions for the channels owned by AB and CD. Transactions for the 4-user channel have black labels, while transactions for the 2-user channels have magenta labels.

An extension of the FFO protocol **[Law22b]** that supports more than two users is used to implement the hierarchical channel's HTLC outputs.

In the 2-user FFO protocol, the offered user can resolve an HTLC in its favor by putting an HTLC-kickoff transaction (that reveals the HTLC's secret) on-chain, waiting for a relative delay of the other user's *to_self_delay* parameter, and then putting an HTLC-success control transaction on-chain that spends the HTLC-kickoff transaction's output. Once the HTLC-success transaction is on-chain, the offered user is guaranteed to be able to resolve the HTLC in their favor (provided the channel's Commitment transaction is still off-chain). In particular, the offered user will be able to put a pre-signed HTLC-payment transaction on-chain that spends the output of the HTLC-success transaction and pays the HTLC funds to the offered user.

On the other hand, if the offered user does not get the HTLC-kickoff transaction on-chain (and thus reveal the HTLC's secret) before the HTLC's expiry, the offering user can prevent the HTLC-success transaction from being put on-chain by spending a conflicting transaction that spends the HTLC-kickoff transaction's output. The offering user is guaranteed to have time to put the conflicting transaction on-chain because of the relative delay between the HTLC-kickoff and HTLC-success transactions.

In order to scale this protocol to a multi-user offered party, the payee is selected as the user that must put the required HTLC-kickoff and HTLC-success transactions on-chain in order for the HTLC to succeed. The output of the HTLC-success transaction requires signatures from **all** of the users in the offered party, so that any one of those users can guarantee that the offered party will get the HTLC payment (provided all users in the offered party have a fully-signed HTLC-payment transaction with sufficient fees). This allows users in the offered party to use a channel protocol to rebalance the funds provided by the HTLC, even if one or more of the users in the offering party remains offline or uncooperative (thus preventing a merging of the HTLC's funds into the offered party's main output).

In order to scale to a multi-user offering party, the relative delay between the HTLC-kickoff and HTLC-success transactions is set to the maximum of the *to_self_delay* parameters within the offering party, and any of the users in the offering party can submit the transaction that conflicts with the HTLC-success transaction. In addition, the transaction that conflicts with the HTLC-success transaction requires signatures from **all** of the users in the offering party in order to guarantee that only transactions with sufficient fees and/or replace-by-fee (RBF) enabled can be submitted. This allows any user in the offering party to force a conflicting transaction to go on-chain if they are willing to pay sufficient fees (provided all users in the offering party have a fully-signed conflicting transaction with sufficient fees). This allows users in the offering party to use a channel protocol to rebalance the HTLC's funds, even if one or more of the users in the offered party remains offline or uncooperative (thus preventing a merging of the HTLC's funds into the offering party's main output).

An example of an HTLC in a 4-user hierarchical channel is shown in Figures 14 and 15 below.

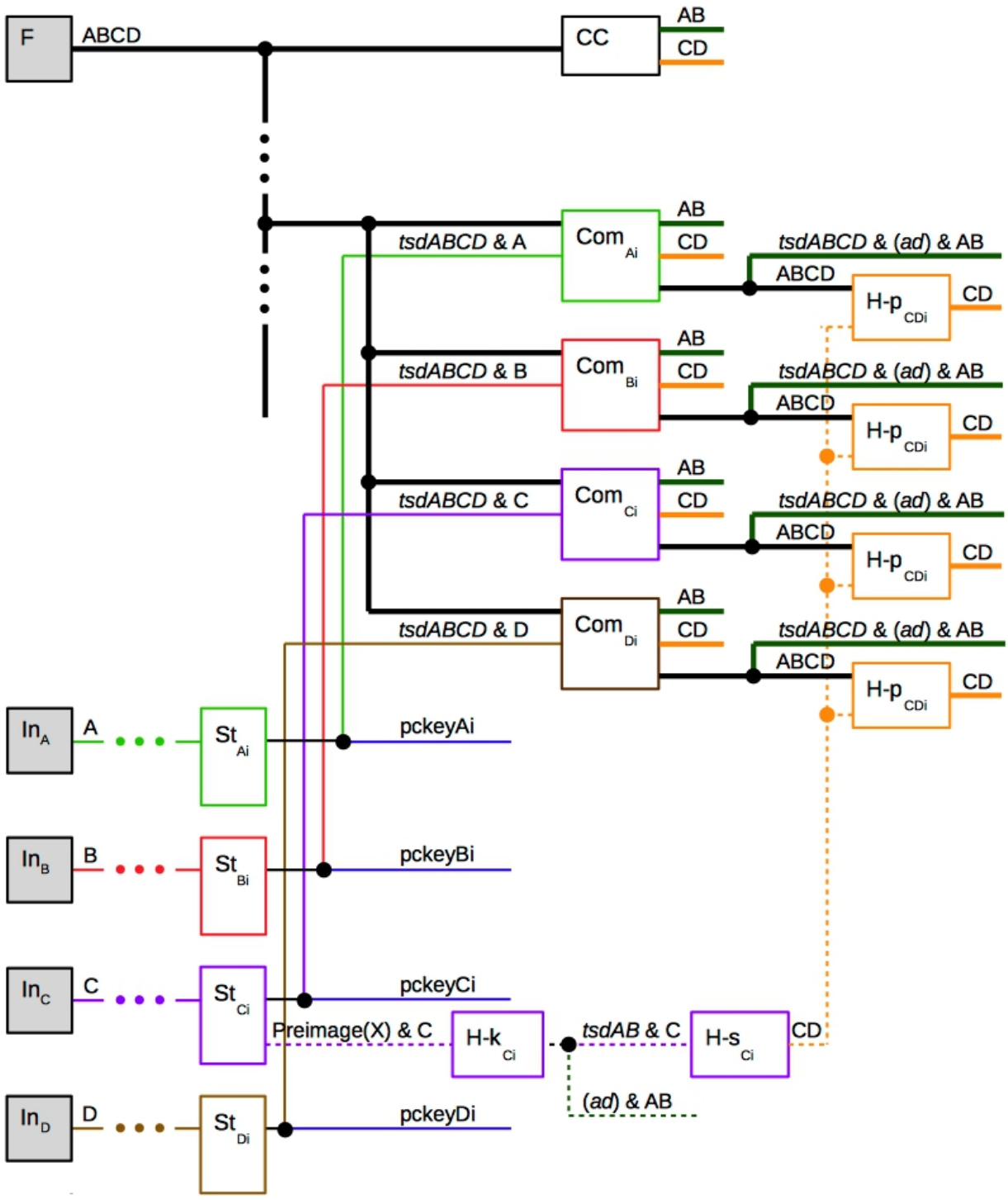


**Figure 14.** 4-user hierarchical channel from Figure 12 showing an HTLC offered by AB (paid by A) to CD (paying to C). The absolute delay "*ad*" equals *eABCD+tsdAB*.

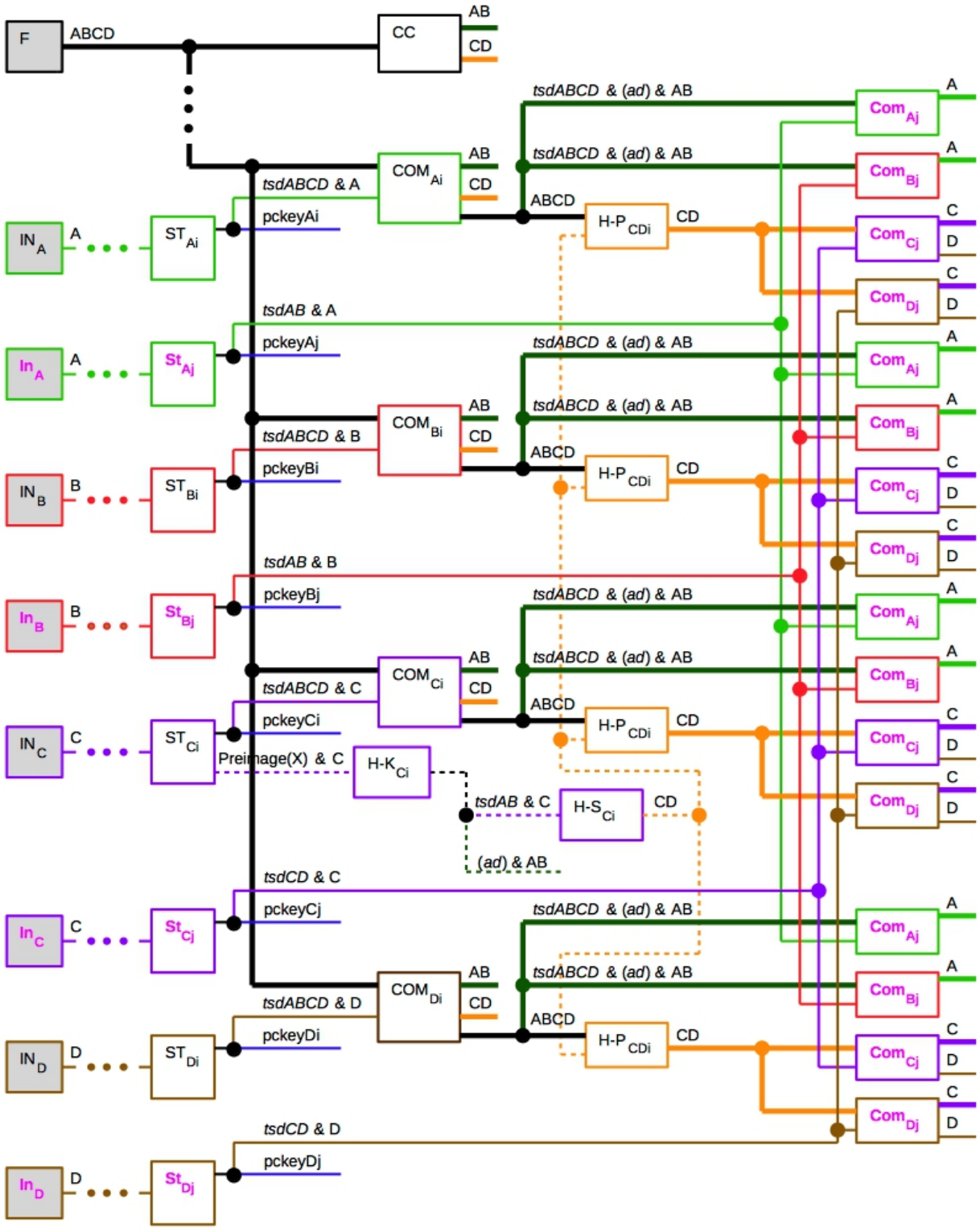


**Figure 15.** 4-user hierarchical channel from Figure 14 showing transactions for the channel funded by the HTLC. If the HTLC succeeds, it pays the value of the HTLC to CD (and then to C with a fee for D). If the HTLC times out, it is refunded to AB (and then to A with no fees being paid). Transactions for channels owned by AB and CD exist but are not shown here.

While Figures 14 and 15 only show transactions for the same state *i*, it is possible that some parties have 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 (as is the case for the FFO protocol **[Law22b]**).

In addition, care must be taken in ordering state updates for a successful HTLC. When state *i* includes an HTLC output and state *i*+1 merges the result of the successful HTLC into the offered party's output, the payee for the HTLC has a state *i* State transaction with a control output for the HTLC that can be spent by an HTLC-kickoff transaction, but the payee's state *i*+1 State transaction does not have a control output for the HTLC. Therefore, the payee must be able to revoke all other users' state *i* State transactions (by knowing their per-commitment keys) before allowing any other user to revoke the payee's state *i* State transaction (by sharing its per-commitment key)[3]. As a result, the payee for the HTLC must be the last user to revoke their state *i* State transaction. An indirect effect of this requirement is that any new state can only merge the value of successful HTLCs into a main output if all of those merged HTLCs have the same payee.

## 5.4 3-User Channels

As was shown in Figures 1 and 4, channels owned by three dedicated users allow a payment to transition between routing through hierarchical channels with more than two users and through 2-user channels. It is straightforward to adapt the protocols for 4-user channels to 3-user channels that consist of a 1-user party and a 2-user party. The key difference is that each output of the 3-user channel that pays to the 1-user party does not fund a new channel, so there is no need for additional control transactions for that output.

Examples of protocols for 3-user hierarchical channels are shown in Figures 16 and 17 below.

3 If the payee did revoke their state *i* State transaction before some other user revoked their state *i* State transaction, that other user would be able to put their state *i* State transaction (and possibly their state *i* Commitment transaction) on-chain, in which case the payee may have to pay a penalty by putting their revoked state *i* State transaction on-chain in order to make the HTLC resolve successfully.

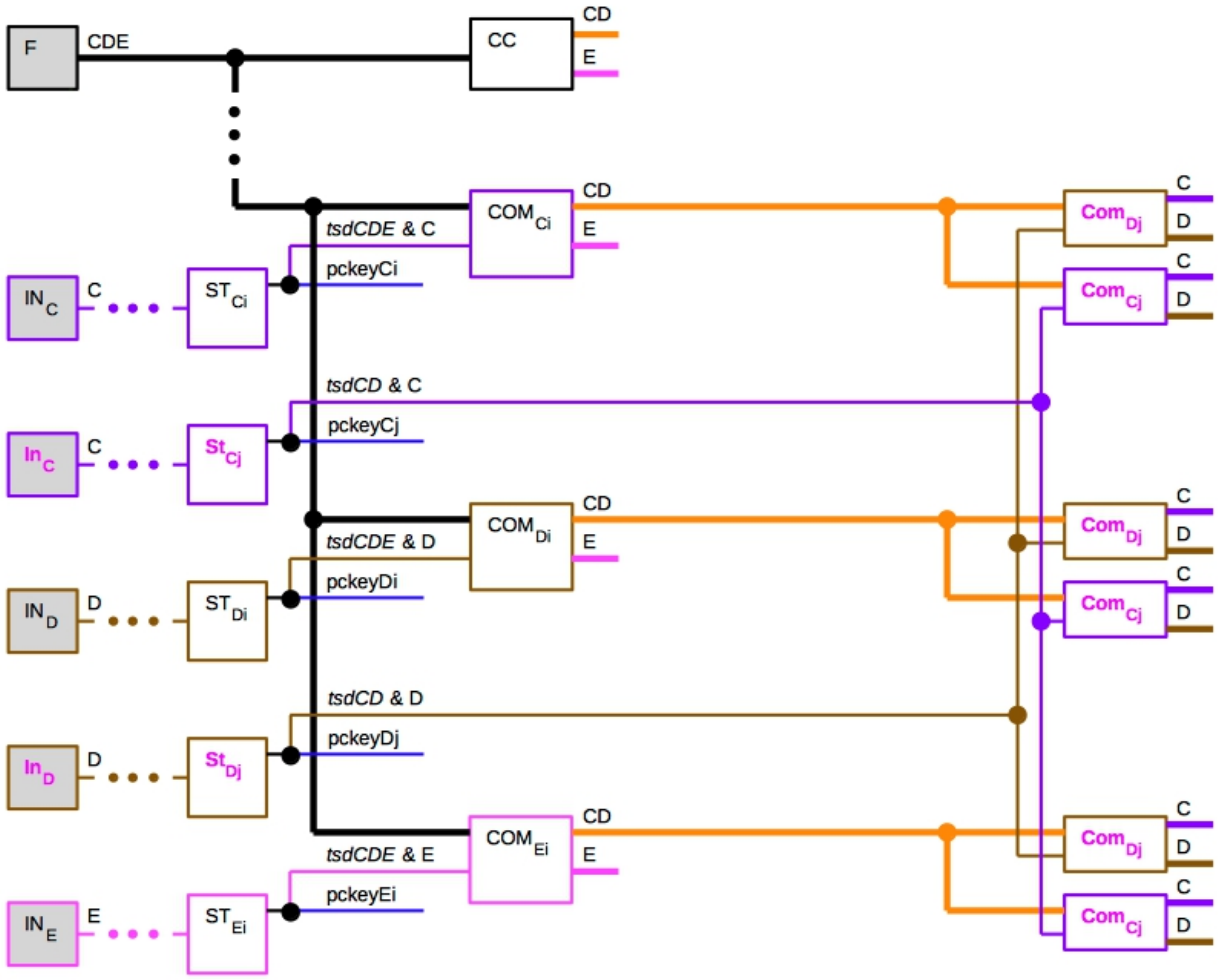


**Figure 16.** 3-user hierarchical channel with outputs to CD and to dedicated user E. The output to CD funds a Lightning channel owned by dedicated users C and D.

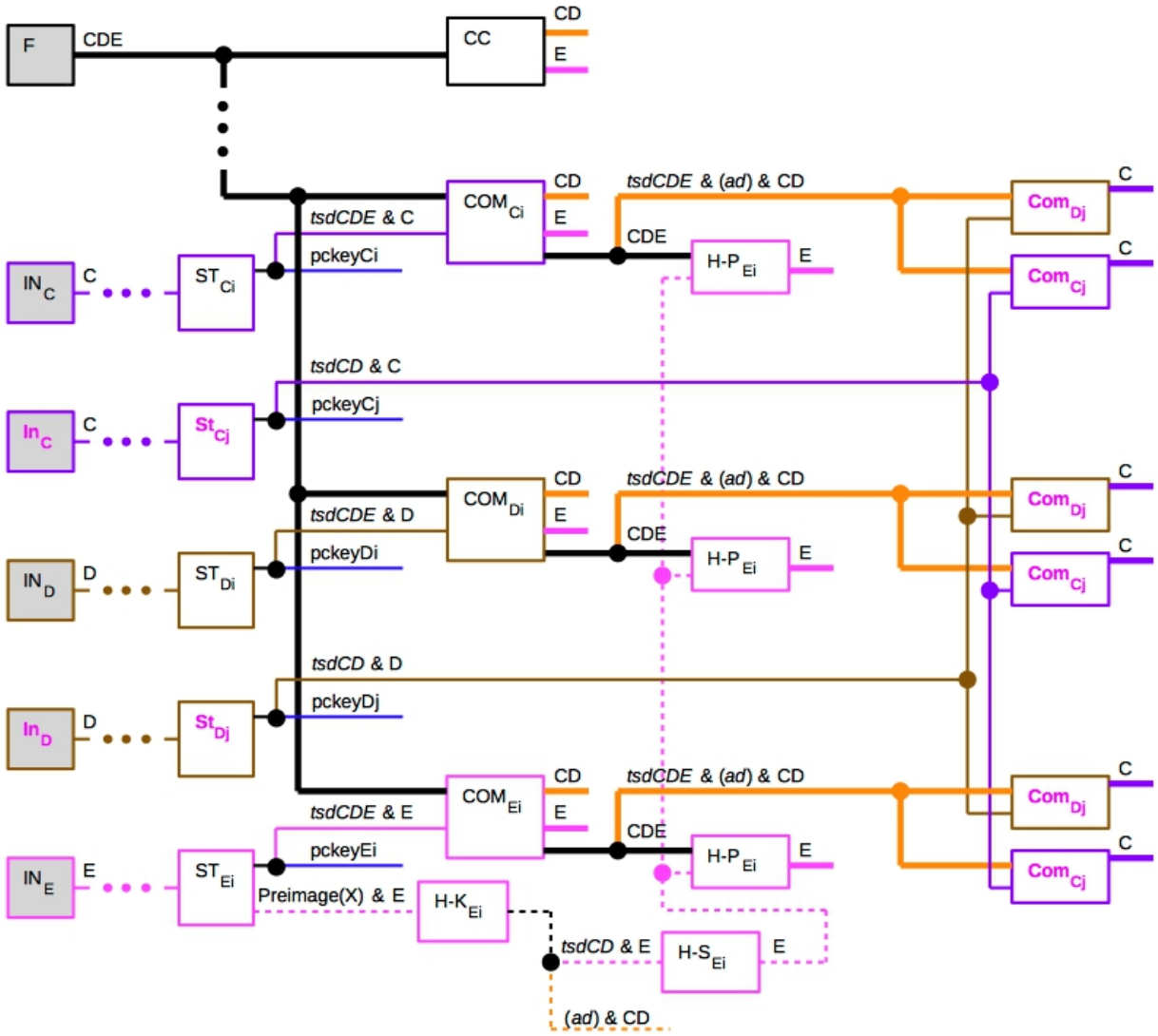


**Figure 17.** 3-user hierarchical channel from Figure 16 showing an HTLC offered by CD (paid by C) to E. If the HTLC succeeds, it pays the value of the HTLC to E (including a fee for E). The absolute delay "*ad*" equals *eCDE+tsdCD*. If the HTLC times out, it is refunded to CD (and then to C, with no fees being paid). Transactions for the channel owned by CD that are not related to the HTLC exist but are not shown here.

## 5.5 Payments to and from Casual Users

Section 4 showed how hierarchical channels can be used to support payments to and from casual users without stranding any channel capacity. In addition to 3-user and 4-user hierarchical routing channels owned by dedicated users, payments to and from casual users require hierarchical payment channels where one party is a casual user and the other party consists of two dedicated users.

As was the case for hierarchical routing channels, the state of a hierarchical payment channel is maintained using a protocol based on the Tunable-Penalty Factory protocol **[Law23]**. However, rather

than allowing each user to put their State transaction on-chain at any time (as in the Tunable-Penalty Factory protocol), the dedicated users have an absolute delay of $T_{BCi}$ before they can submit their State transactions for state $i$, where $T_{BCi}$ is casual user A's *to_self_delay* in the future when A signs B's and C's Commitment transactions for state $i$. The purpose of this absolute delay will be explained below.

An example of a hierarchical payment channel with a casual user and two dedicated users is shown in Figure 18 below.

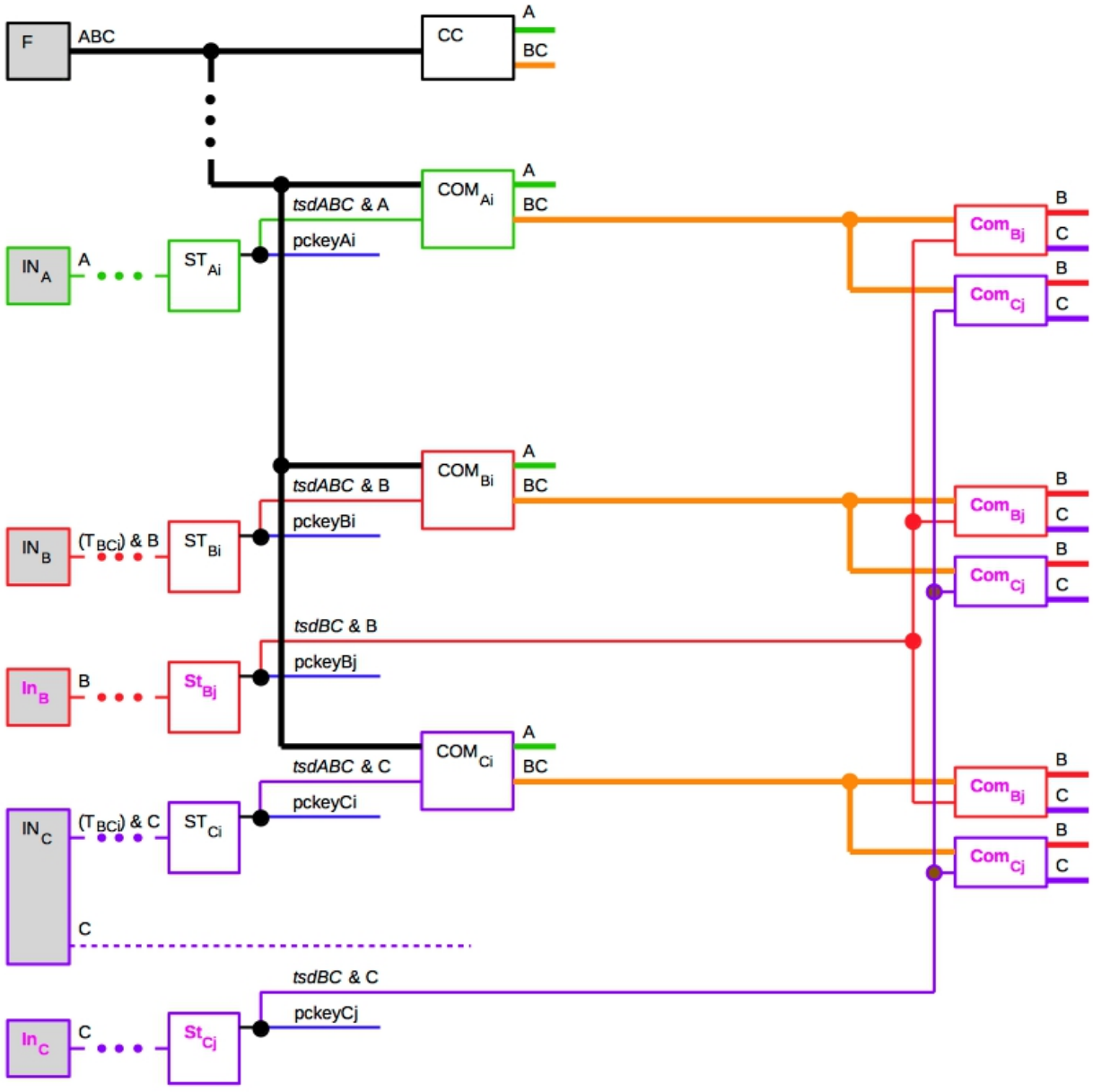


**Figure 18.** 3-user payment channel with outputs to casual user A and to BC. The output to BC funds a routing channel owned by dedicated users B and C. Transactions for the routing channel owned by B and C have magenta labels. The second output of C's Individual transaction is unused here.

Unlike hierarchical routing channels (which use an extension of the FFO protocol for HTLCs), hierarchical payment channels for casual users implement HTLCs using a protocol that is an extension of the FFO-WF protocol **[Law22b]** to more than two users.

In the 2-user FFO-WF protocol, only the casual user's Commitment transaction has any HTLC outputs. Instead, the dedicated user's Commitment transaction pays the full value of all outstanding HTLCs to the dedicated user. An absolute delay on the dedicated user's State transaction prevents the dedicated

user from putting their Commitment transaction on-chain unless the two users fail to update the channel's state to reflect the success or failure of the outstanding HTLCs, and the casual user fails to put their State and Commitment transactions on-chain. Thus, the dedicated user's Commitment transaction acts as an incentive for the casual to put their State and Commitment transactions on-chain if the HTLCs are not resolved off-chain.

When the FFO-WF protocol is extended to three users, only the casual user's Commitment transaction has any HTLC outputs, both dedicated users' Commitment transactions pay the full value of all outstanding HTLCs to them, and both of the dedicated users' Commitment transactions have an absolute delay (denoted $T_{BCi}$) before they can be put on-chain.

## Payments from casual users

First, consider payments from casual users.

In the 2-user FFO-WF protocol **[Law22b]**, once the casual user has put their State transaction on-chain, the dedicated user has a relative delay of their *to_self_delay* parameter in which to put an HTLC-success transaction (that reveals the HTLC's secret) on-chain. If the dedicated user puts this HTLC-success transaction on-chain promptly, they are guaranteed to be able to receive the HTLC's funds. On the other hand, if the dedicated user does not put the HTLC-success transaction on-chain during the allotted relative delay, the casual user can put a transaction that conflicts with the HTLC-success transaction on-chain, thus guaranteeing that the HTLC's funds will be refunded to the casual user.

In order to scale to a multi-user offered party, the relative delay between the casual user's State transaction and the casual user's transaction that conflicts with the HTLC-success transaction is set to the maximum of the *to_self_delay* parameters within the offered party. In addition, the HTLC-success transaction requires signatures from **all** of the users in the offered party, so that any one of those users can guarantee that the offered party will get the HTLC payment (provided all users in the offered party know the HTLC's secret and have fully-signed HTLC-success and HTLC-payment transactions with sufficient fees). This allows users in the offered party to use a channel protocol to rebalance the funds provided by the HTLC, even if the casual user remains offline or uncooperative (thus preventing the HTLC-success transaction from being put on-chain). This functionality is important to avoid stranding capital, as the casual user may remain offline for months without violating the channel protocol.

An example of a payment from a casual user is shown in Figure 19 below.

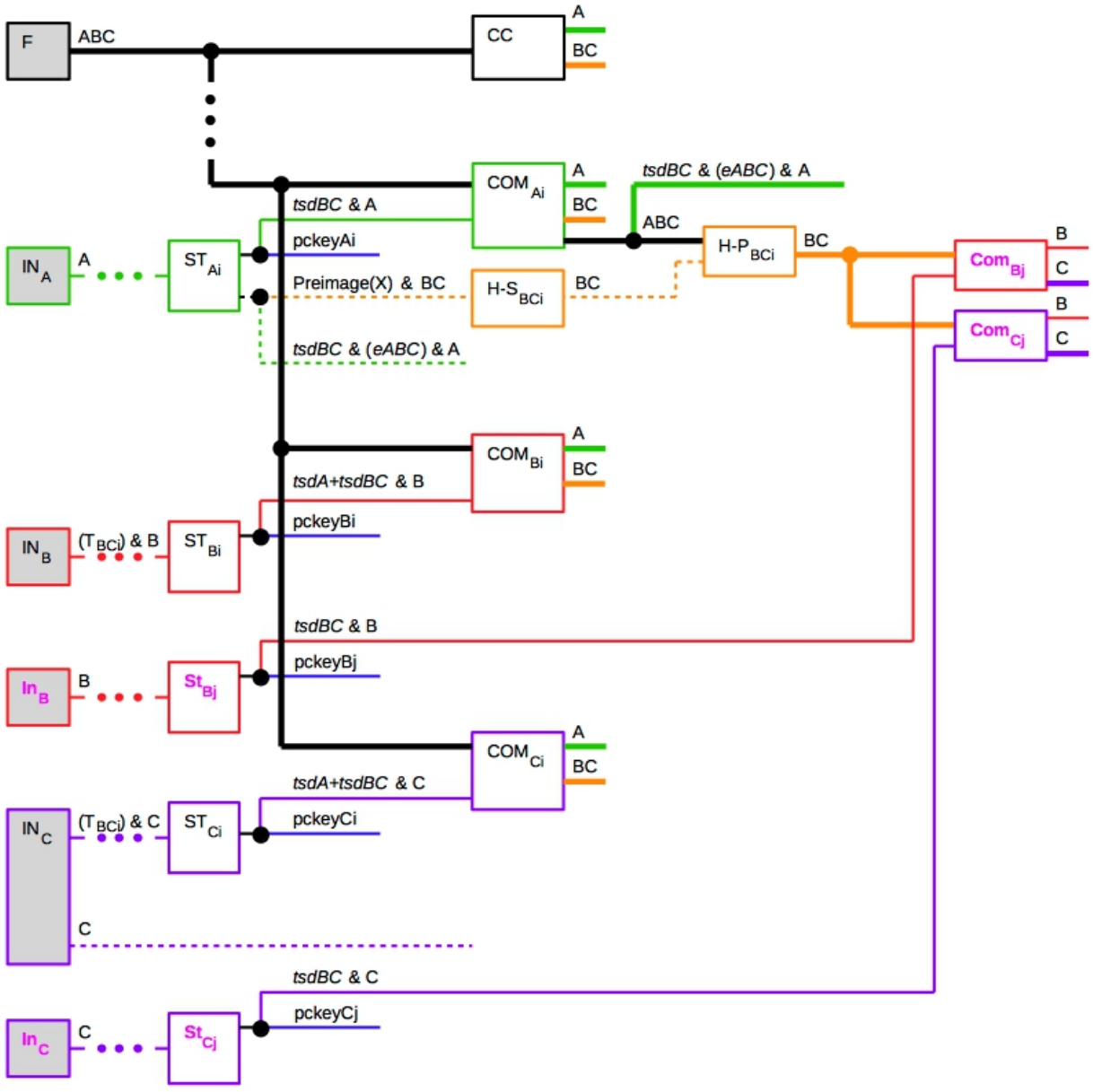


**Figure 19.** 3-user channel from Figure 18 sending a payment from casual user A. A's payment is held in the HTLC output that pays to BC (and then to C along with a fee to B) if C provides the required preimage, or is refunded to A otherwise. There are also transactions for the routing channel owned by BC, but they are omitted from this figure for clarity.

Note that in Figure 19, dedicated user C can share Preimage(X) with B so that B can guarantee that H-$P_{BCi}$, or $COM_{Bi}$ or $COM_{Ci}$ can be put on-chain. As a result, B and C can each guarantee that the funds from this HTLC will pay to them, thus allowing B and C to use those funds to route payments as soon as Preimage(X) is revealed, even if A stays offline for multiple months.

## Payments to casual users

Next, consider payments to casual users.

In the 2-user FFO-WF protocol **[Law22b]**, once the HTLC's expiry has passed, the dedicated user can put an HTLC-timeout transaction on-chain that guarantees that the HTLC's funds will be refunded to the dedicated user (provided the casual user's Commitment transaction is not yet on-chain). On the other hand, if the casual user puts their State transaction and HTLC-success transaction (which reveals the HTLC's secret) on-chain prior to the HTLC's expiry, the dedicated user's HTLC-timeout transaction cannot be put on-chain (as it conflicts with the HTLC-success transaction), thus guaranteeing that the casual user can obtain the HTLC's payment.

In order to scale to a multi-user offering party, the payee is selected as the user that can put the HTLC-timeout transaction on-chain. The output of the HTLC-timeout transaction requires signatures from **all** of the users in the offering party, so that any one of those users can guarantee that the offering party will get the HTLC refund (provided all users in the offering party have a fully-signed HTLC-refund transaction with sufficient fees). This allows users in the offering party to use a channel protocol to rebalance the funds refunded by the HTLC, even if the casual user remains offline or uncooperative (thus preventing a merging of the HTLC's refund into the offering party's main output of the casual user's Commitment transaction).

As in the 2-user FFO-WF protocol **[Law22b]**, whenever state $i$ includes an HTLC offered to the casual user, state $i$+1 does not include any HTLCs offered to the casual user. This constraint can be met by having the casual user reveal the secrets for all outstanding HTLCs offered to them[4], thus allowing those HTLCs to be removed from the next channel state.

An example of a payment to a casual user is shown in Figure 20 below.

4 This is possible because the casual user is the destination for the payments corresponding to those HTLCs, so the casual user knows the HTLCs' secrets.

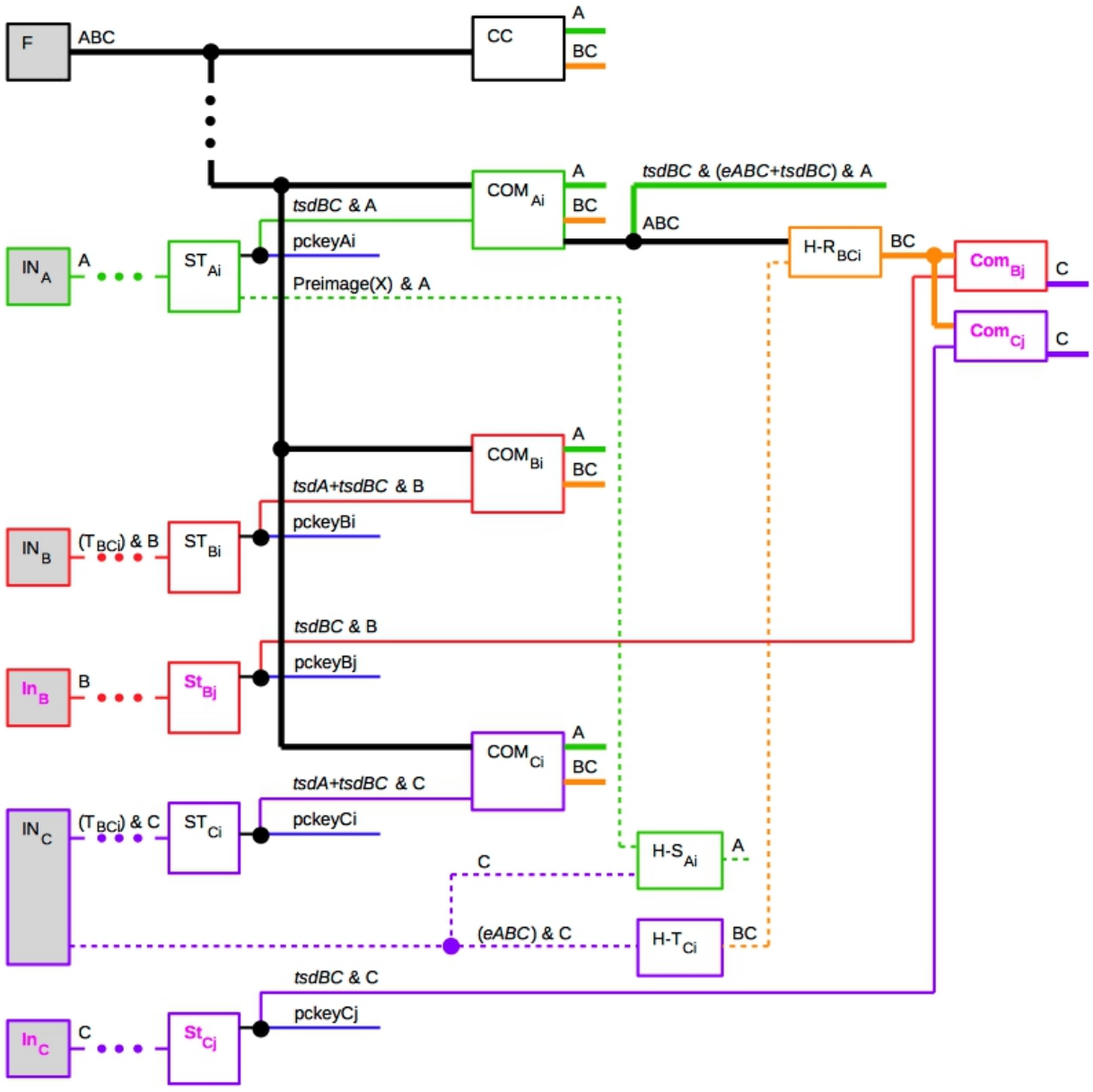


**Figure 20.** 3-user channel from Figure 18 receiving a payment to casual user A. The payment to A is held in the HTLC output that pays to A if A provides the required preimage, or is refunded to BC (and then to C) otherwise. There are also transactions for the routing channel owned by BC as shown in Figure 18, but they are omitted from this figure for clarity.

# 6 Discussion

## 6.1 Hierarchical Channels with Three or More Levels

The examples shown above include hierarchical channels with one or two levels. However, as is clear from the definition of a hierarchical channel, it is possible to create a hierarchical channel with three or more levels. A hierarchical channel with three or more levels has added flexibility, as the capacity of any 2-level (or 1-level) hierarchical channel that it creates can be increased or decreased off-chain.

While it is not clear if this amount of flexibility is required, there are situations where it could be valuable. For example, consider a 4-user hierarchical channel with parties AB and CD. If this 4-user channel is funded by an on-chain UTXO, the channel owned by AB could have its capacity reduced by making a payment like the one shown in Figure 1, without affecting the capacity of the logical channel owned by CD. Alternatively, by swapping the roles of A and B with those of C and D, the channel owned by CD could have its capacity reduced without affecting the capacity of the logical channel owned by AB. However, it would not be possible to simultaneously set the capacity of the logical channels owned by AB and by CD to zero. On the other hand, if the 4-user channel owned by A, B, C and D were funded by a 3-level hierarchical channel (and if the other physical channels owned by AB and by CD were funded off-chain by hierarchical channels), it would be possible to set the capacity of the logical channels owned by AB and by CD to zero.

Increasing the number of levels in a hierarchical channel has some disadvantages. In particular, funding a channel with an off-chain UTXO from an N-user hierarchical channel requires that N sets of off-chain copies of the channel must be maintained, one per each of the N Commitment transactions that could fund it. This multiplicative factor in off-chain transactions, and in the number of signatures that must be exchanged to update those transactions, could become prohibitive quite quickly.

## 6.2 Hierarchical Channels vs. Channel Factories

As has been demonstrated, hierarchical channels provide much more flexibility than channel factories in resizing channels off-chain. This additional flexibility comes from the fact that hierarchical channels support 2-party HTLCs, thus allowing updates within one hierarchical channel to be tied atomically to updates in other (potentially hierarchical) channels.

However, even more flexibility could be obtained by adding HTLCs to factories that allow arbitrary outputs. Specifically, hierarchical channels force the users to form two fixed parties, with the hierarchical channel only having outputs for those two parties (plus HTLC outputs, each of which resolves to paying one or the other of those parties). Instead, one could use a factory protocol (such as the Invalidation Tree channel protocol **[DW15]** or the Tunable-Penalty Factory protocol **[Law23]**) to create outputs for arbitrary subsets of users and to create HTLCs between arbitrary subsets of users.

The ability to create an HTLC between arbitrary subsets of users may enable shorter routing paths for certain Lightning payments. However, the ability to route between arbitrary subsets of users within a

multi-user factory is difficult to represent in the Lightning channel graph. Furthermore, the creation of a new HTLC between arbitrary subsets of users requires that all users in the factory sign the new HTLC, so a single user's unavailability would prevent the HTLC from being created. As a result of these challenges, staying within the limitations imposed by hierarchical channels seems reasonable.

### 6.3 Software Engineering Considerations

As was noted in Section 5.1, a key challenge in creating protocols for hierarchical channels is the requirement that the protocol be able to support more than two users. It was noted that any channel factory protocol, including the Invalidation Tree channel protocol **[DW15]**, the Tunable-Penalty Factory protocol **[Law23]** or the Single-Commitment Factory protocol **[Law23]** could be used to implement the hierarchical channel's state. In addition to having the best performance for an on-chain unilateral close, the Tunable-Penalty Factory protocol is also attractive from a software engineering perspective. This is because the structure of the Tunable-Penalty Factory protocol is very similar to the FFO and FFO-WF channel protocols **[Law22b]**, extensions of which are used to implement HTLCs in hierarchical channels (see Sections 5.3, 5.4 and 5.5). As a result, it should be possible to create a single code base that supports both efficient 2-user channels and hierarchical channels with more than two users.

## 7 Related Work

A number of previously-published protocols allow Lightning channels to be resized off-chain.

In particular, channel factories **[BDW18]** based on the Invalidation Tree protocol **[DW15]**, the eltoo protocol **[DRO18]**, the Tunable-Penalty Factory protocol **[Law23]** and the Single-Commitment Factory protocol **[Law23]** all allow channels to be resized off-chain. In addition, CoinPools **[NR]** extend the channel factory functionality by allowing arbitrary unilateral withdrawals from the pool while leaving the remaining users in a single pool. These protocols differ from hierarchical channels by forcing all channels (or user-owned outputs) that lose capacity off-chain to be balanced by other channels (or user-owned outputs) within the same factory or pool that gain capacity off-chain. As a result of this constraint, there is a desire to scale channel factories and CoinPools to large numbers of users. Unfortunately, such scaling is limited by their requirement for high interactivity **[Ria22]**, especially if one or more casual users are included. In contrast, hierarchical channels with just three or four users are able to resize channels very flexibly off-chain. In addition, eltoo and CoinPools differ in that they require a change to the underlying Bitcoin protocol.

Finally, hierarchical channels make critical use of Russell's proposal **[Rus22]** for *channel_update_v2* messages.

# 8 Conclusions

Sending bitcoin between users can be performed on-chain. However, allocating bitcoin to a Lightning channel allows it to be sent off-chain nearly instantly, with much lower fees, and in a far more scalable manner.

Similarly, Lightning channel capacity can be moved to where it is needed on-chain. However, allocating Lighting channel capacity within a hierarchical channel allows the channel to be resized off-chain nearly instantly, with much lower fees, and in a far more scalable manner.

In addition, hierarchical channels can be used to support trust-free, watchtower-free casual users without stranding any Lightning channel capacity. As a result, providing watchtower-freedom to casual users appears to be cost-effective.

Given that these results can be achieved without making changes to the underlying Bitcoin protocol, it is hoped that hierarchical channels will eventually be adopted in BOLTs **[BOLT]** and implemented in order to improve the scalability, efficiency, and usability of the Lightning network.

# Acknowledgments

Thanks to Anthony Towns for his idea of including an absolute delay in the Commitment transactions' HTLC outputs, thus eliminating the need for a *max_cltv_expiry* parameter in the FFO and FFO-WF protocols and in the protocols for hierarchical channels.

# References

**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.

**DRO18** Christian Decker, Rusty Russell and Olaoluwa Osuntokun. eltoo: A Simple Layer2 Protocol for Bitcoin. 2018. See https://blockstream.com/eltoo.pdf.

**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.

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

**Law22b** John Law. Factory-Optimized Channel Protocols For Lightning. See https://github.com/JohnLaw2/ln-factory-optimized.

**Law23** John Law. Efficient Factories For Lightning Channels. See https://github.com/JohnLaw2/ln-efficient-factories.

**NR** Gleb Naumenko and Antoine Riard. CoinPools. See https://coinpool.dev.

**Ria22** Antoine Riard. Conjectures on solving the high interactivity issue in payment pools and channel factories. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-April/020370.html.

**Rus22** Rusty Russell. [RFC] Lightning gossip alternative. See https://lists.linuxfoundation.org/pipermail/lightning-dev/2022-February/003470.html.

**Tei22** Bastien Teinturier. Re: Watchtower-Free Lightning Channels For Casual Users. See https://lists.linuxfoundation.org/pipermail/lightning-dev/2022-October/003712.html.