Nitro Protocol

Tom Close

November 14, 2021

Abstract

State channels are an important technique for scaling blockchains, allowing a fixed set of participants to jointly run an application in order to determine how a set of assets should be distributed between them. In this paper, we present a new protocol for constructing state channel networks, allowing state channels to be opened and closed without on-chain transactions and decreasing the number of deposits that need to be held. The protocol readily extends to n-party channels and we include the construction of a 3-party virtual channel.

1 Motivation

State channels are an important technique for scaling blockchains. In a state channel, a fixed set of participants execute a series of state transitions off-chain, in order to determine how a set of assets should be distributed between them. By allowing participants to execute these transitions off-chain, the state channel removes load from the blockchain, allowing it to support the same level of activity with fewer transactions.

The allowed state transitions are specified by a set of update rules, which can be thought of as defining an application that runs in the state channel. When running an application in this collaborative manner, participants need guarantees that the application will not stall indefinitely and that the transition rules will be respected. State channels provide these guarantees by providing a challenge mechanism, whereby participants can appeal to the blockchain to enforce these conditions. By harnessing the blockchain, state channels become a trustless execution environment for running multi-party applications.

The final state of the channel is used to determine how a set of assets should be distributed. In order to ensure that these assets can only be distributed according to the outcome of the channel, they must be held in escrow for the duration of the channel. These assets are often referred to as the state deposit for the channel. By ensuring that the *state deposit* is fully governed by the channel's outcome, state channels also provide a form of instant finality; once the outcome of the channel is known the participants can consider the value to have been distributed knowing that they now have the capability to claim the assets on-chain at a future point of time of their choosing.

In the simplest case, a separate state desposit is required for each new channel. For each application a set of participants wish to run, at least one party needs to perform an on-chain transaction to transfer assets into the state deposit, and each time it is closed at least one participant must perform an on-chain transaction to claim their share. This limits the effectiveness of state channels as a scaling solution, making it only suitable for the case where a large number of transactions are executed between a single group of participants. We refer to these naive channels as *direct channels*, as they are supported directly by funds held on the blockchain.

State channel networks move beyond this limitation, using new types of channels to break the direct link between state deposits and channels. Ledger channels allow one state deposit between a fixed set of participants to support multiple simultaneous applications between that set of participants. Virtual channels take this one step further by allowing state deposits with a shared intermediary to support applications between a set of participants who have no state deposit between themselves.

1.1 Our contribution

In this paper we present Nitro Protocol, a protocol for constructing state channel networks. We give a detailed description of how to construct ledger channels and virtual channels, including how to safely open and close these channels entirely off-chain. Taken with our earlier work on ForceMove [**?**], this paper gives a complete specification for building a state channel network capable of running arbitrary state channel applications.

Our work is unique in that the channels in our networks are both homogeneous and independent. By homogeneous, we mean that channels function exactly the same whether they are direct channels or whether they are funded via a ledger channel or virtual channel; in particular, there are no time limits or other restrictions placed on the applications that run in virtual channels. Channels can even transition from virtual channels to direct channels while an application is running, without affecting the application's execution in any way. By independent, we mean that updates to different channels are unrelated. We have no way of applying atomic updates across multiple channels, nor do we have need to do this. The independence of channel updates makes it far easier to reason about the incentives involved in designing state channel applications, as with independence you can reason about whether updates will be accepted on a per channel basis.

All the work in this paper is generalizable to n-party channels and, by way of example, we include the first explicit construction of a virtual channel between 3 parties with a shared intermediary. Other examples include utility protocols for tasks such as top-ups and partial withdrawals from ledger channels.

We also develop a set of tools for reasoning about the correctness of the protocols we present.

2 Existing work

There are many examples of state channels and off-chain scaling projects. In this section we limit ourselves to a review of published work on the subject of off-chain payment and state channel networks.

The Lightning network [**?**], which went live in March 2018, provides off-chain payments for the Bitcoin blockchain. The payments make use of hashed time-locked contracts (HTLCs), which can be thought of as payments that are conditional on a hash pre-image being revealed before a given point in time. This construction allows payments to be routed through an arbitrary number of intermediaries but is strictly limited to payments. The Raiden network [**?**] provides the same functionality for the Ethereum blockchain and launched on the mainnet in December 2018.

Celer Network [**?**] proposes a state channel construction that extends HTLCs to allow payments that are conditional on the outcome of an arbitrary calculation. The outcome of the calculation can specify the amount of funds that move, as well as whether the payments should go through at all. The paper gives a high-level justification of how the construction yields state channels capable of running arbitrary state machine transitions.

Perun [**?**, **?**] proposed a different flavour of state channel construction, viewing state channels as a direct interaction between two parties instead of a series of conditional payments. This makes it very clear that state channel updates themselves need only be shared between the participants in the channel, and do not need to be routed through a network. The authors specify a virtual channel construction, allowing two-party channels to be supported through intermediaries, and prove its correctness using the UC framework. The proof relies on the fact that their virtual channels have a pre-determined validity time, after which the channel must be settled.

Counterfactual [**?**] gives a state channel construction using the technique of counterfactual instantiation, a form of logic that reasons about constructions that could be deployed to the chain if required. The channels they describe are n-party and they give a high-level overview of how to construct ‘meta-channels’ that allow channels to be supported through intermediaries. While the paper itself does not specify the details of how to construct meta-channels, many of these details can be found in their publicly released source code.

3 Modelling State Channel Networks

In this section, we present a simple model for state channel networks. The model is intended to be easy to understand and reason about, while still capturing the essential features. It will form the basis for the protocols introduced later in the paper, as well as for the tools used to prove the correctness of those protocols.

3.1 A System of Balances

At the heart of our model lies a simple system of balances. We start by describing that system.

In this paper, we simplify the explanation by only considering a single asset, which we will refer to as **coins**. We will further simplify matters by specifying that quantities of coins will have no maximum size, taking values in $\mathbb{Z}^+$. This allows us to avoid dealing with integer overflows when presenting operations. These simplifications do not cause any limitations in practice and all the work here can be applied to state channel networks that manage an arbitrary number of asset types with maximum values.

In order to store value, a state channel network must be backed by assets held on-chain. In our explanation, we assume that these funds are held and managed by a single[1] smart contract, which we will refer to as the **adjudicator**.

The first purpose of the adjudicator is to store the balance of coins held for a given address. Addresses can correspond either to participants in the network or to state channels. A **participant address** is a regular blockchain address, generated in the standard way from the signature scheme. A **channel address** is formed by taking the hash of the participant addresses along with a nonce, k, that is chosen by the participants in order to distinguish their channels from one another. Given an address, A, we assume that the properties of the signature scheme and hashing algorithm make it impossible to find a private key for A or to construct a channel whose address is A, if they are not previously known.

We model the adjudicator as having a simple mapping that stores the quantity of coins for an address. If an address does not appear in the table we take the balance to be zero. Figure 1 introduces the notation we will use to describe the system.

The **deposit** operation, $D_A(x)$, is an on-chain operation used to increase A's balance by x coins. There are no restrictions on who can deposit coins for an address, but the transaction must always include a transfer of x coins into the adjudicator.

The **withdrawal** operation, $W_A(x)$, can be used to withdraw coins held at participant address, A, by any party with the knowledge of the corresponding

[1]We assume this for simplicity only - in practice the functionality could be split across multiple contracts.

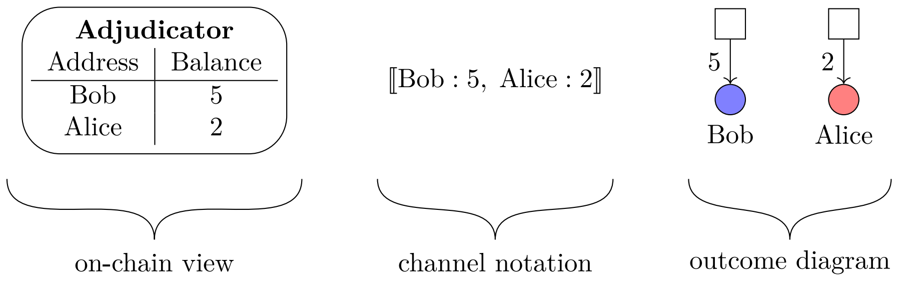


Figure 1: *Three different ways of representing the situation where Bob has* 5 *coins stored against his address in the adjudicator and Alice has* 2*. The on-chain view shows a pictorial representation of the state in the adjudicator. Channel notation is useful for writing the state in equations and will later be extended to cover off-chain state as well. In the outcome diagram, the white squares represent adjudicator balances and the solid circles are coloured to represent the different participants.*

private key. In practice, the withdrawal should also specify the blockchain address where the funds should be sent. A potential method signature is `withdraw(fromAddr, toAddr, amount, signature)`, where `signature` is A's signature of the other parameters[2].

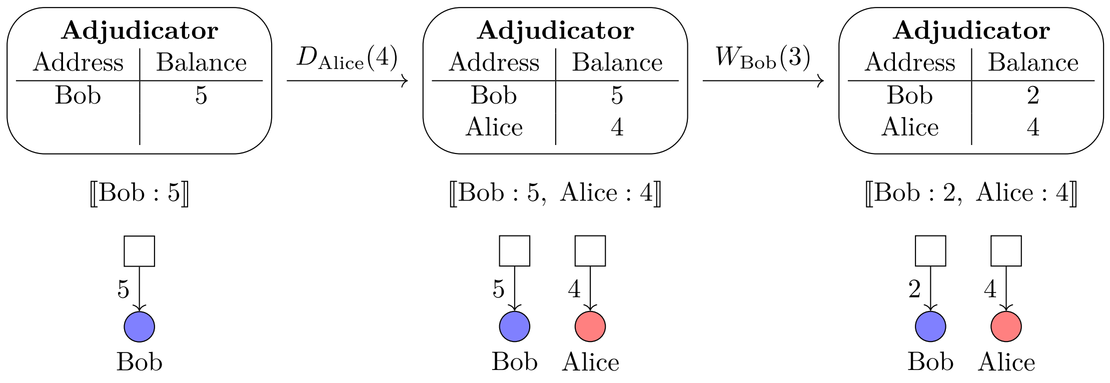


Figure 2: *Deposits and withdrawals. The deposit can be called by any blockchain user, provided that it is accompanied with a transfer of the same number of coins into the adjudicator. For the withdrawal to be successful, the withdrawal parameters must be signed with Bob's private key.*

To recap, we now have a simple smart contract that can store a balance against an address, which either represents a participant or a channel. The balances

[2]In practice, we could add the `senderAddress` to the parameters to sign, in order to prevent replay attacks by other parties.

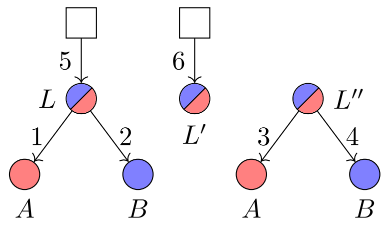
can be increased and decreased through deposits and withdrawals. Anyone can deposit into an address of either type[3] but funds can only be withdrawn from participant addresses - and only by a party who knows the private key. The total coins held by the smart contract should always equal the sum of the balances.

3.2 State Channel Outcomes

In our model, a state channel is an off-chain protocol followed by a set of participants, enabling them to reach an **outcome** that can be used to update the balances on the chain. The format and interpretation of the outcome is specified by the state channel network protocol being used. An example of a type of outcome is the *allocation*, which is used in both Turbo and Nitro protocol, and which consists of a list of recipient addresses and totals that specify how the channel's balance should be distributed.

Adjudicator

Address	Balance	Outcome
L	5	A: 1, B: 2
L'	6	
L''		A: 3, B: 4



$$[\![\mathrm{L} : 5 \mapsto (A : 1, B : 2),\ L' : 6,\ L'' \mapsto (A : 3, B : 4)]\!]$$

Figure 3: *Representation of (allocation) outcomes in the three different diagram formats. We show the three possible cases: a channel, L, with both a balance and an outcome; a channel, L', with a balance but no outcome; and a channel, L'', with an outcome but no balance. We represent the channels with split circles, coloured to represent the participants of the channel, which we have taken to be A and B.*

Central to the approach of the model is to split the updating of the balances into two steps:

1. **Finalization**: getting to a point where the outcome of the channel is stored on the blockchain.
2. **Redistribution**: updating the balances in the adjudicator according to that outcome.

The bulk of this paper is focussed on the redistribution step, which we will discuss further in section 3.3.

[3]But there is nothing to be gained from depositing into a participant address.

Understanding how an outcome is finalized inevitably involves understanding the rules of operation of the state channel: from exchanging states to launching and responding to challenges. The ForceMove protocol completely specifies these rules of operation, in a way that is compatible with the work in this paper.

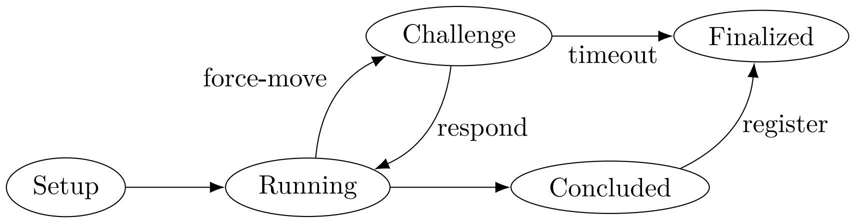


Figure 4: *ForceMove Channel Operation. There are two ways for an outcome to become finalized: (i) through a challenge that times out before anyone responds, and (ii) through the registration of a conclusion proof.*

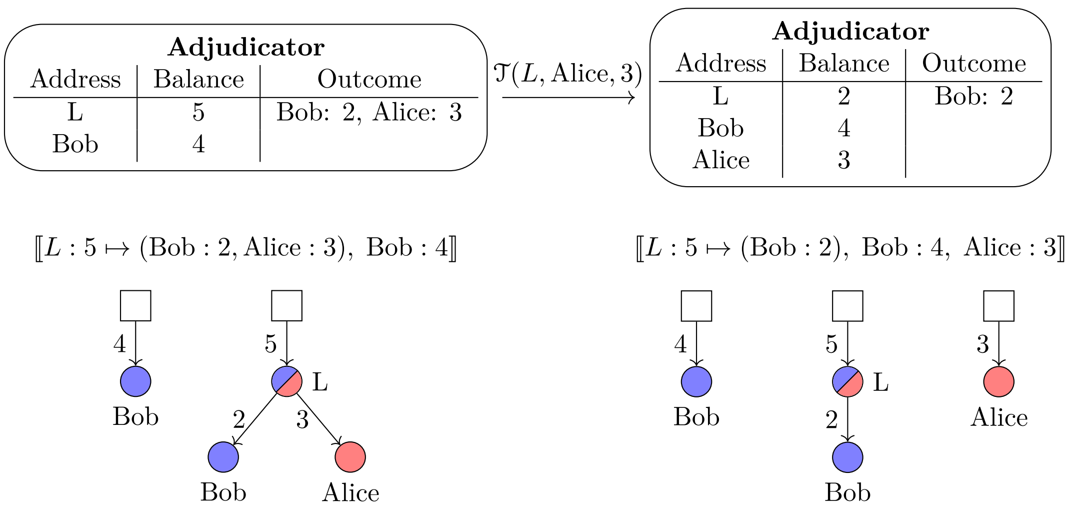
In ForceMove, there are two ways for an outcome to be finalized. The first corresponds to a non-collaborative closing of the channel: one participant launches an on-chain challenge, starting a timeout period; if no other participant responds during the timeout period, then the challenge times out and the outcome corresponding to the challenge state is finalized. The second corresponds to a collaborative closing of the channel: all the participants sign a special conclude state, called a conclusion proof; any participant can then register this outcome on-chain to create a finalized outcome. By closing the channel collaboratively the participants avoid having to wait for the challenge period.

For the purpose of the model, it is crucial that the rules of operation only allow one outcome to be finalized for each channel. As we will see in section 4, it is also important that the rules of operation make it possible to know which outcome(s) a participant can finalize from a given state.

3.3 Redistribution

The second part of extracting the funds from a state channel is the redistribution step. Redistribution involves calling a sequence of on-chain **operations** to manipulate the balances and finalized outcomes. The allowed operations are defined by the network protocol used. Figure 5 shows an example of the transfer operation from Turbo protocol.

The operations change the state of the adjudicator: typically both balances and outcomes. There is no restriction on who can trigger the operations. In this paper, we present all operations separately and assume they are called separately. In practice, we expect that implementations would provide some utility methods that combine common sequences of operations, to improve gas efficiency.



Adjudicator

Address	Balance	Outcome
L	5	Bob: 2, Alice: 3
Bob	4	

$\xrightarrow{\mathfrak{T}(L, \text{Alice}, 3)}$

Adjudicator

Address	Balance	Outcome
L	2	Bob: 2
Bob	4	
Alice	3	

$[\![L : 5 \mapsto (\text{Bob} : 2, \text{Alice} : 3),\ \text{Bob} : 4]\!]$

$[\![L : 5 \mapsto (\text{Bob} : 2),\ \text{Bob} : 4,\ \text{Alice} : 3]\!]$

Figure 5: *A example of a redistribution operation from Turbo protocol. Here the transfer operation is used to move* 3 *coins out of channel L to Alice. Note: the parts of the adjudicator responsible for storing challenges are omitted from the diagram.*

To recap, we now have a system where participants can deposit into state channel addresses on-chain. By running a state channel, participants can reach an outcome and ensure this outcome is finalized on-chain. Once the outcome is finalized, the funds held in the channel can be redistributed to other participants and channels by calling on-chain operations. Participants can then withdraw any funds that have been redistributed to their address.

In the next section, we will use this model to develop some tools for constructing state channel networks and proving their safety. In particular, we will develop the logic which allows us to use the fact that a given outcome could be finalized if necessary, to avoid putting that outcome on-chain at all.

4 Reasoning about State Channels

In this section, we outline our approach to proving the correctness of our state channel network constructions.

The very nature of state channels tends to make the logic complex. In a state channel, value is moved between participants by exchanging commitments about the distribution of assets held on-chain. Inevitably you end up reasoning about the commitments you hold and their interpretation by the chain, which necessarily also includes reasoning about the possible actions of the other parties both internal and external to the channel.

On top of this, there is an inherent danger in entering into a state channel relationship, as it requires funds to be locked on-chain. In order to be safe, protocols need to be robust against other parties acting maliciously and/or ceasing to cooperate at any point in the protocol. We need to show that at any point, any participant can extract the value currently owed to them in spite of any actions taken by other parties.

4.1 Channel Funding and Value

We will start by considering the interpretation of the outcome of a state channel. Suppose A is a participant in a state channel, L, that reaches an (allocation) outcome, ω, that allocates x coins to A. What does that mean for A? In particular, how much more can A withdraw from the system due to that outcome?

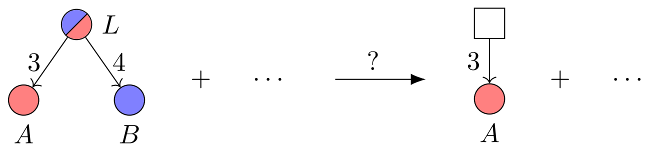


Figure 6: *Understanding whether a channel is funded amounts to understanding how much value can be extracted from the system when that channel reaches an outcome.*

There is one case where the answer to these questions is very straightforward: where the channel L itself has enough coins in the adjudicator to cover the entire allocation. In this case, we say the channel is **directly funded**. If this happens, A will receive all x coins allocated to them in ω.

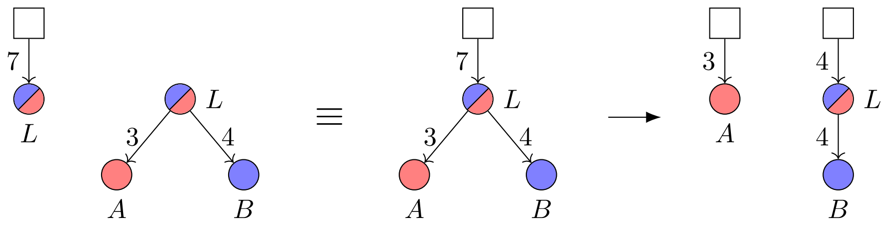


Figure 7: *Direct funding. One case where we know we can extract the full value allocated to us in a channel outcome, is when the channel has a sufficient balance in the adjudicator to cover the full allocation.*

This is a good start, but the whole point of state channel *networks* is to move beyond the case where every channel needs to be directly funded. Suppose instead that L is not directly funded but there is another channel, L', that is. Further suppose that L' has reached an outcome where all its coins are allocated to L. Using this outcome, we know we can redistribute the coins in the adjudicator to L, recreating the situation above, where L was directly funded. Therefore, in this situation we also know that A will receive the x coins from the outcome of L, and that L can be considered to be **indirectly funded**. Note that we did not actually need to perform the redistribution on-chain to reach this conclusion - we just needed to be able to reason that the outcome enabled us to.

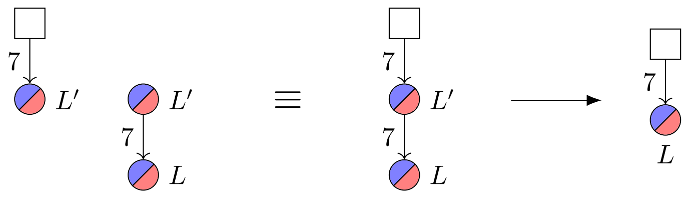


Figure 8: *Indirect funding. If we possess another outcome, that allocates funds to L, we know we can convert this to a situation where L is directly funded. We can therefore consider L to be indirectly funded.*

In the previous paragraph, we looked at the case where L' had already reached an outcome. In general, this will not be the case; the power of state channels comes from the ability to move between many potential outcomes as the interaction progresses. In order to say whether a channel is funded, we will need to start not with an outcome but with a **network state**. The network state, Σ, for a participant, A, consists of:

1. The state of the adjudicator:
   (a) The balances held
   (b) Any finalized outcomes
2. For each channel A is a participant of:
   (a) Signed commitments that A has received
   (b) Signed commitments that A has sent
   (c) Private information held by A

The private information always includes A's signing key for the channel and can also include information specific to the application running in the channel; for example, in a game of battleships the private information would include the positions of A's ships. Note that A's network state does not include a detailed model of which commitments are held by specific other participants - just what A has sent and received. Generally we assume that all other participants are controlled by a single adversary, pooling their resources and commitments.

We can now proceed with some definitions of funding and value:

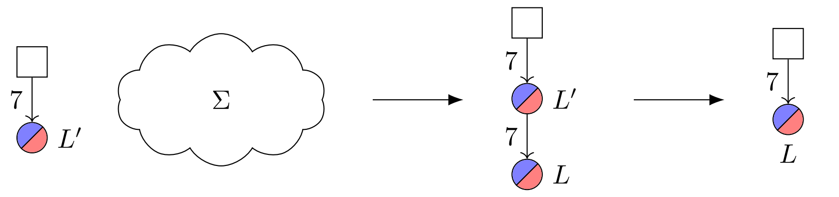


Figure 9: *In practice, we deal with a network state, Σ, and not in definite outcomes. To understand value, we also need to be able to reason about which outcome(s) could result from the current network state, as well as the value those outcomes then deliver.*

A channel χ is **funded** for participant A with x coins, if A has an unbeatable strategy for obtaining a state where χ is directly funded with x coins.

The **value** for participant A of a network state Σ is the maximum x for which A has an unbeatable strategy for obtaining a state where the balance of A's address in the adjudicator is x coins.

The concept of an unbeatable strategy can involve a full range of actions allowed within the protocol including signing commitments, refusing to sign commitments, launching/responding to on-chain challenges and calling on-chain operations to redistribute funds. We will cover this in more detail in section 4.3.

4.2 Network Constructions

Now that we have defined what we mean by a channel being funded and a state having value, we can start to talk about the state channel network constructions that will form the bulk of the paper. A construction specifies both the network state for each participant and a sequence of states that can be used to reach it. Presenting a construction will follow the same rough pattern:

1. Show that a given network state funds a channel.
2. Show it can be built from a known state, using a sequence of value-preserving single channel updates.

The single channel update requirement is a key decision in the design the protocol. This means that we do not allow atomic updates across multiple channels; each update to the system comprises sending or receiving a single statement applying to just one channel. This keeps channel updates independent, which makes it a lot easier to reason about finalizability on a per-channel basis.

We require that the sequence of state transitions is value preserving for each participants involved. While the power of state channel networks comes from being able to move value off-chain, opening and closing channels can be viewed as rewriting the existing state in a different form and therefore should not change the value. We furthermore make the assumption that participants will be willing to make any transition that preserves their value, meaning that value-preservation is both a necessary and sufficient property for constructing network states. We call this last assumption the **Simple Transition Rule**.

In the case where we ignore the cost of the on-chain redistribution operations, the simple transition rule is straightforward and non-controversial. If we consider this cost, the situation becomes a bit more subtle, as moving from a simpler to a more complicated construction actually leads to a slight decrease the value that is extractable from the network. In practice, using the simple transition rule means we are assuming that the utility from being able to fund channels off-chain will outweigh the slight increase in cost in the worst-case scenario. Modelling the cost of the on-chain operations is beyond the scope of this paper.

4.3 Unbeatable Strategies

In the definitions of value and funding, we talked about having an unbeatable strategy for obtaining some state on-chain. The means that whatever actions (or lack of actions) other participants and external parties might take, the target state is still obtainable. This is not the easiest definition to work with: to show that a strategy is unbeatable it seems that you have to consider all possible actions other parties could take. In this section, we will break this down and give some tools to make it easier to show that a strategy is unbeatable.

We start by outlining the rules for interacting with the blockchain. When evaluating whether a strategy is unbeatable, we make the following assumptions about blockchain transactions:

1. **Transactions are unimpeded**: given that the current time is t and $\epsilon > 0$, then it is possible for any party to apply any operation, O, on-chain before time $t + \epsilon$.
2. **Transactions *can* be front-run**: given two parties, p_1 and p_2, and two operations, O_1 and O_2, there is no way for p_1 to ensure that they can apply O_1 to the chain before p_2 applies O_2.

The first assumption sidesteps issues of censorship, chain congestion and timing considerations around the creation of blocks. In practice, this assumption should hold if ϵ is sufficiently large, which can be accomplished by picking sensible channel timeouts. The second assumption rules out any strategies that rely on executing a given transaction on-chain before someone else executes a different one.

We now take the task of constructing an unbeatable strategy and break it into two stages: finalization and redistribution.

Finalization happens on a per-channel basis, with different channels finalizing independently. This makes it easier to reason about which outcomes are possible. In general, we cannot assume that the outcome will be known; we might have to take multiple possible outcomes through to the redistribution step. The

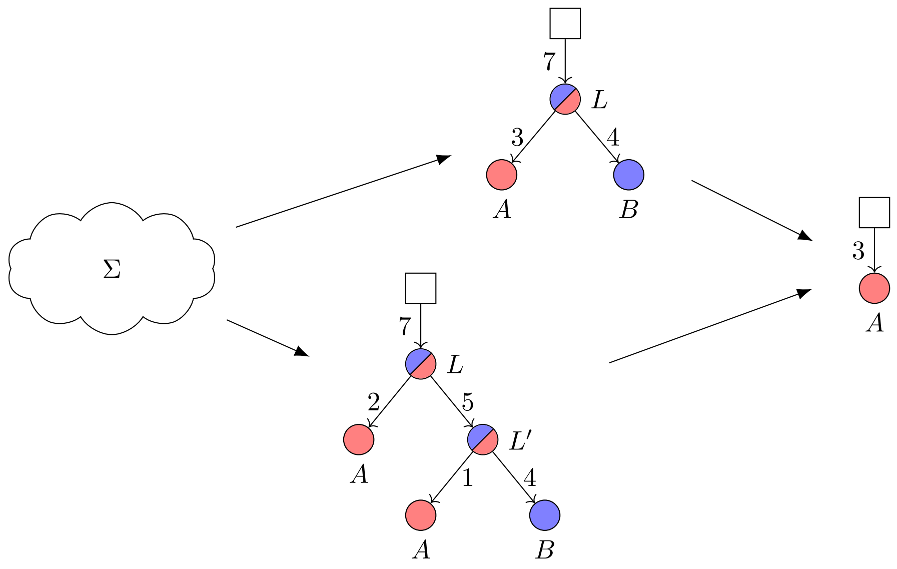


Figure 10: *When calculating value, we will often need to consider all outcomes that are possible from a given network state and show that they all allow us to extract the same value from the network.*

finalization step depends heavily on the rules of the state channel framework. We will cover finalization in more detail in section 4.4.

Reasoning about when a redistribution strategy is unbeatable, depends heavily on the protocol involved. We will cover the logic here in the sections on Turbo and Nitro protocol. In Turbo, it turns out that the answer is simple: any strategy that works is unbeatable. In Nitro, it is more complicated to show that redistribution strategies are unbeatable but we provide a few tools to help.

4.4 Finalizable Outcomes

We say an outcome, Ω, is **finalizable** for participant A, if A has an unbeatable strategy for finalizing this outcome in the adjudicator. We use the notation $[\chi \mapsto \Omega]_A$, to represent a state of a channel, χ, where the outcome, Ω, is finalizable by A.

$$[\chi \mapsto \Omega]_A \xrightarrow{\text{A's unbeatable strategy}} [\![\chi \mapsto \Omega]\!] \tag{1}$$

It follows from the definition that exactly one of the following statements is true about a channel χ at any point in time:

1. Finalized outcome: the outcome of χ has already been finalized on-chain: $[\![\chi \mapsto \Omega]\!]$
2. One participant has multiple finalizable outcomes: participant p has one or more finalizable outcome(s), $\Omega_1, \ldots, \Omega_m$, and no other participant has any finalizable outcomes. We write this $[\chi \mapsto \Omega_1, \ldots, \Omega_m]_p$.
3. Multiple participants share one finalizable outcome: there are at least two participants, $P = \{p_1, \ldots, p_m\}$, who share the same finalizable outcome, Ω. We write this $[\chi \mapsto \Omega]_{p_1,\ldots,p_m}$.
4. No finalizable outcomes: there are no participants with any finalizable outcomes.

The definition of finalizability excludes the case where two different finalizable outcomes are held by different participants, as in this case at least one participant's strategy would be beatable by the other participant's strategy. None of the protocols we present make use of the last case, where no participant has a finalizable outcome.

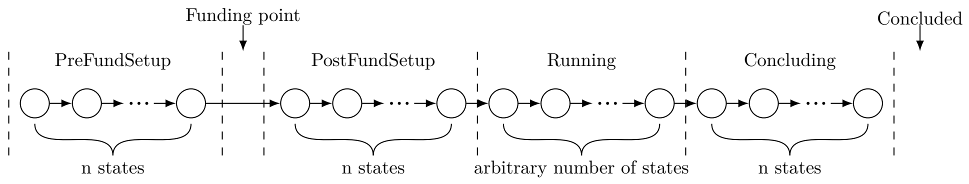


Figure 11: *Every ForceMove channel has at least two points when the outcome is universally finalizable: one at the funding point and one when the channel has concluded. This is important when reasoning about creating state channel network constructions.*

In the special case where the outcome of a channel is finalizable by all its participants, we say that the outcome is **universally finalizable**. For a ForceMove channel, this happens at the following points in its lifecycle:

1. After the first n states have been broadcast. In this state, we say the channel is at the **funding point**.
2. When a single conclusion proof is known to each participant. In this state, we say the channel is in the **concluded state**.

It is an important property of ForceMove that all channels have one universally finalizable state at the beginning of their lifecycle and one at the end[4].

If a participant has no finalizable outcomes, their analysis of the network needs to be performed in terms of their **enabled outcomes**. The enabled outcomes for a participant, p, is defined as the set of outcomes that p has no strategy

[4]If a channel does not end with a conclusion proof, it ends with an expired on-chain challenge, in which case the outcome is already finalized on-chain.

to prevent from being finalized. We write the set of enabled outcomes for p as $[\chi \mapsto \Omega_1 \ldots \Omega_m]_{(p)}$.

For any participant, p, in a channel, χ, exactly one of the following statements is true at a given point in time:

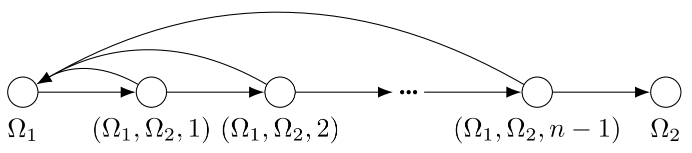
1. p has at least one finalizable outcome.
2. p has at least two enabled outcomes.

Note that if a participant has only enabled a single outcome, that outcome must be finalizable for them.

4.5 Consensus Game

Another important example of universally finalizable states comes from the **consensus game**. The consensus game is a ForceMove *application*, which means it specifies a certain set of transitions rules that can be used to define the allowed state transitions for a ForceMove channel. We will make heavy use of the consensus game throughout the paper.

The consensus game provides a way for participants to move from one universally finalizable outcome to another, provided that they all agree. The participants start in a state where Ω_1 is the universally finalizable outcome. One participant proposes the new outcome, Ω_2. On their turn, each subsequent participant decides whether to accept the transition to the new outcome or whether to cancel the transition and return to Ω_1.



Ω_1
$(\Omega_1, \Omega_2, 1)$
$(\Omega_1, \Omega_2, 2)$
$(\Omega_1, \Omega_2, n-1)$
Ω_2

Figure 12: *A consensus game transition from Ω_1 to Ω_2, for a channel with n participants. The counter records how many participants have approved the transition. If all participants agree, they finish in a state with outcome Ω_2. Any participant can reject the transition, returning to the state with Ω_1.*

Throughout the only enabled outcomes for any participant are Ω_1 and Ω_2. In particular, a participant has the finalizable outcome $[\chi \mapsto \Omega_1]_p$ until they approve the transition, and then enabled outcomes $[\chi \mapsto \Omega_1, \Omega_2]_{(p)}$ until they receive the final state. When the final state is broadcast, every participant has the finalizable outcome $[\chi \mapsto \Omega_2]_p$.

4.6 Outcomes First

In practice, it is hard to write networks states down concisely. Instead, we will write our constructions in terms of outcomes and use the properties of the consensus game to reason that (a) network states exist that lead to this outcome and (b) we can find a sequence of network states to transition from one outcome to another.

In particular, we will present sequences of sets of outcomes, where each set differs only in the outcome of a single consensus game channel. Each of the outcomes will have the same value to all participants. We then know that, using the properties of the consensus game, we can transition between these two states with a consensus game transition, without enabling any additional outcomes.

To show that we can build a construction, it is therefore sufficient to present the sequence of sets of equal-value outcomes, where each set differs only in the outcome of a single consensus game channel. This is the approach we will take in the rest of the paper.

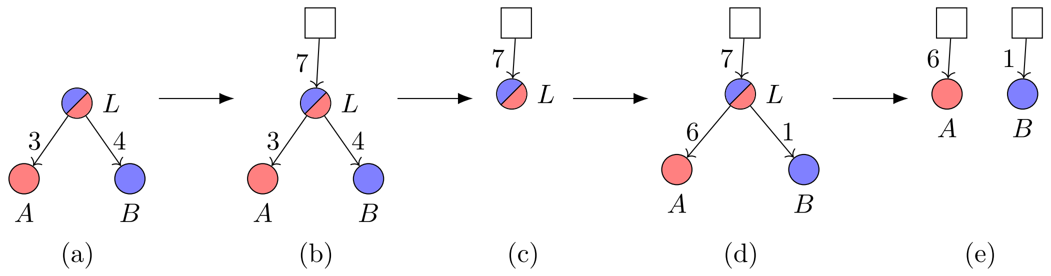


Figure 13: *In (a), A and B have exchanged the first two states in the channel L, bringing them to the funding point. At this point the channel is not yet funded. In step (b), both participants have deposited into the adjudicator. In step (c), the channel L is running. A and B do not have a finalizable outcome and the ultimate outcome is governed by the rules given by the channel's game library. In step (d), A and B have created a conclusion proof and therefore have another universally finalizable outcome. They are then able to finalize this outcome on-chain and withdraw their funds in (e).*

5 Turbo Protocol

Turbo protocol has one type of outcome (the allocation) and one on-chain operation (the transfer). You will already be somewhat familiar with these, as they formed the basis of the examples in sections 3 and 4. In this section we will make these more precise, present the related result on distribution and give some example constructions to cover common tasks such as opening and closing sub-channels.

5.1 Allocations and Transfer

An **allocation** is a list of pairs of addresses and totals, $(a_1{:}v_1, \ldots, a_m{:}v_m)$, where each total, v_i, represents that quantity of coins due to each address, a_i. We assume that each address only appears once in the allocation and require that implementations enforce this by ignoring any additional entries for a given address after the first.

The allocation is in priority order, so that if the channel does not hold enough funds to pay all the coins that are due, then the addresses at the beginning of the allocation will receive funds first. We say that 'A **can afford** x for B', if B would receive at least x coins, were the coins currently held by A to be paid out in priority order.

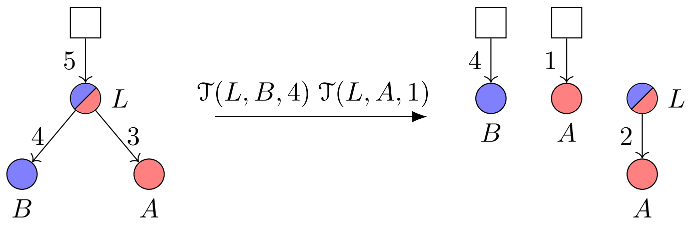


Figure 14: *Allocations pay out in priority order. In the diagram, B is drawn to the left of A to show that B has higher priority in the outcome of L. In this example, L can afford 4 coins for B, but can only afford 1 coin for A.*

Turbo introduces the **transfer** operation, $\mathcal{T}(A, B, x)$, to trigger the on-chain transfer of funds according to an allocation. If A can afford x for B, then $\mathcal{T}(A, B, x)$:

1. Reduces the funds held in channel A by x.
2. Increases the funds held by B by x.
3. Reduces the amount owed to B in the outcome of A by x.

If A cannot afford x for B, then $\mathcal{T}(A, B, x)$ fails, leaving the on-chain state unchanged.

5.2 Unbeatable Redistribution

As we mentioned in section 4.3, reasoning about redistribution is easy in Turbo: if you can find one strategy to move a certain amount into an address, then no-one else can prevent this from occurring. In this section we will justify this by presenting an algorithm for calculating the funding for each address.

We will restrict ourselves to looking at strategies and counter-strategies involving transfer operations only, ignoring deposits and withdrawals. Deposits and withdrawals cannot be required as part of a strategy and cannot help as part of a counter-strategy. The intuition here is that a deposit into the system cannot reduce the value of any address and cannot increase the value of any address by more than the value of the deposit. Withdrawals can only occur from participant addresses and are the only way funds can leave these addresses, so cannot affect values elsewhere.

We will only consider the case where the network of outcomes forms a directed acyclic graph (DAG), where the nodes are channels and the edges represent allocations from one channel to the other. While it is technically possible to create outcomes with cycles, it is also possible for any participant in the channel to prevent this from happening. We therefore consider non-DAG outcome networks to be outside the scope of the protocol.

We commence our value calculation by taking a *topological ordering* of the nodes of the graph. A topological ordering is an ordering of nodes such that, if $N_1 \mapsto N_2$ is an edge, then $N_1 < N_2$. It is a known result that all DAGs have at least one topological ordering.

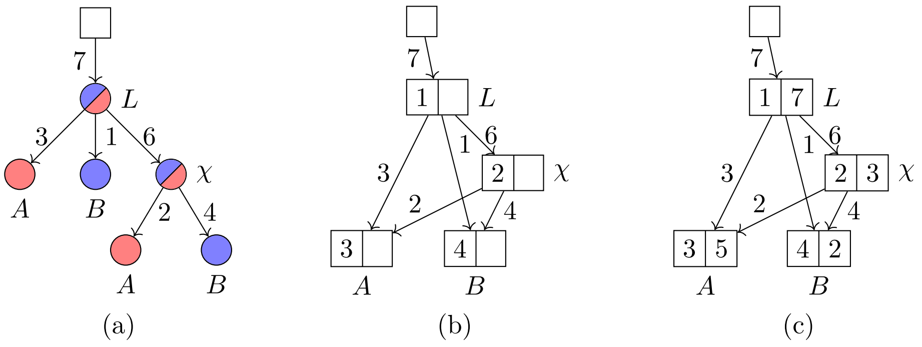


Figure 15: *Diagram (a) shows the outcome network that is the input to the value calculation. In (b), we have reformulated (a) as a DAG with uniquely labelled nodes by merging the two A and B nodes. We have also labelled the nodes with a topological ordering. In (c), we have completed the algorithm giving each node its funding/value.*

Turbo Value Algorithm

1. Choose a topological ordering, `OrderedNodes`, for the network.
2. Create a mapping, `Values`, from `OrderedNodes` to $\mathbb{Z}^+$. Initialize this mapping by setting `Values`$[n]$ to be the balance held for n in the adjudicator, for each $n \in$ `OrderedNodes` (with `Values`$[n] = 0$ if n's address does not appear).
3. For each $n \in$ `OrderedNodes`$[n]$ (taken in order):
   (a) Let `remainingFunds` $=$ `Values`$[n]$.

(b) For each (`destinationNode`, `payout`) in n's allocation (taken in order):

   i. Let $x = \min(\texttt{payout}, \texttt{remainingFunds})$.

   ii. Increase $\texttt{Values}[\texttt{destinationNode}]$ by x.

   iii. Decrease `remainingFunds` by x.

4. Then $\texttt{Values}[n]$ gives the value of node n.

It is not hard to see that it is impossible to find a strategy that gives any node a value higher than allocated by this algorithm. It is also not hard to construct a strategy for a node to obtain the value allocated by the algorithm, if necessary by actually implementing the algorithm up until that node. Given that we are only considering counter-strategies with transfers, and we have done every possible transfer on these channels, we know that there are no transfers that can interrupt this algorithm. It is also easy to see that calling transfers out of order does not affect the ultimate result.

In Turbo, it is therefore easy to calculate the value of each node and find unbeatable strategies for extracting the value of that address.

5.3 Ledger Channels

A **ledger** channel is a channel whose sole purpose is to provide funding to other channels. We call the channels that are funded by the ledger channel **sub-channels** of the ledger channel. All ledger channels run the consensus game.

Although this has already been covered, for completeness we will quickly recap how a sub-channel can be considered to be funded by a ledger channel. For example, consider the following setup where a ledger channel, L, allocates the funds it holds to participants A and B and channel $\mathcal{X}$:

$$[\![L : 10]\!], [L \mapsto (A : 3, B : 1, \mathcal{X} : 6)]_{A,B} \tag{2}$$

In this example, $\mathcal{X}$ is funded with 6 coins by L for both A and B. To show this, we have to have an unbeatable strategy for moving to a situation where $\mathcal{X}$ is directly funded with 6 coins. To do this we first note that the outcome $(A : 3, B : 1, \mathcal{X} : 6)$ is finalizable for both A and B, so we can start our strategy by putting this outcome on-chain. Once it is on-chain, the transfer operation $\mathcal{T}(L, \mathcal{X}, 6)$ is all that is required to make $\mathcal{X}$ directly funded. From the Turbo redistribution result, we know that this redistribution strategy is unbeatable.

Note that offloading $\mathcal{X}$ like this should be seen as an action of last-resort, as after the off-load all sub-channels supported by L must be closed on-chain. It is in the interest of both participants to open and close sub-channels collaboratively. We next give some examples to show how this can be accomplished safely.

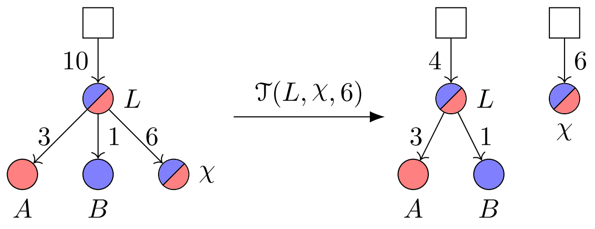


Figure 16: *Offloading a ledger channel. The transfer operation is used to move funds out of the ledger channel L into channel χ, so that χ becomes directly funded.*

5.4 Example Constructions

We now give some examples of how to work with ledger channels on Turbo. We have chosen to present examples that demonstrate the key principles instead of presenting general protocols, as we believe that, once seen, these protocols are easy to extend to the general case.

5.4.1 Opening a Sub-channel

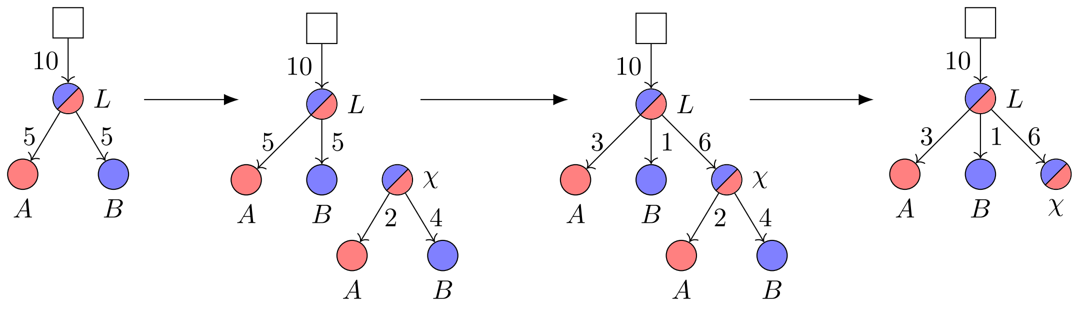


Figure 17: *Opening a sub-channel.*

The utility of a ledger channel derives from the ability to open and close sub-channels without on-chain operations. Here we show how to open a sub-channel.

1. Start in a state where A and B have a funded ledger channel, L, open:

$$[\![L : x]\!],\ [L \mapsto (A : a, B : b)]_{A,B} \tag{3}$$

2. A and B prepare their sub-channel χ and progress it to the funding point. With $a' \leq a$ and $b' \leq b$:

$$[\chi \mapsto (A : a', B : b')]_{A,B} \tag{4}$$

3. Update the ledger channel to fund the sub-channel:

$$[L \mapsto (A : a - a', B : b - b', \chi : a' + b')]_{A,B} \tag{5}$$

5.4.2 Closing a Sub-channel

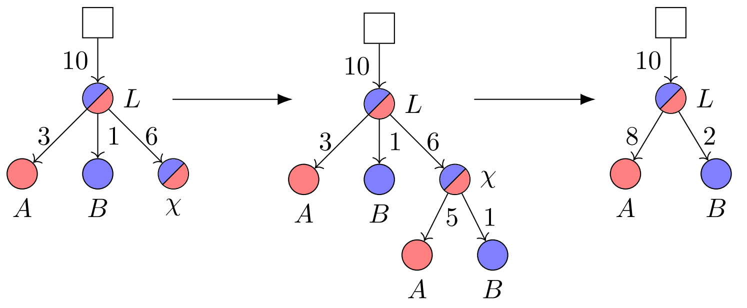


Figure 18: *Closing a sub-channel.*

When the interaction in a sub-channel, χ, has finished we need a safe way to update the ledger channels to incorporate the outcome. This allows the sub-channel to be defunded and closed off-chain.

1. We start in the state where χ is funded via the ledger channel, L. With $x = a + b + c$:

$$[\![L : x]\!],\ [L \mapsto (A : a, B : b, \chi : c)]_{A,B} \tag{6}$$

2. The next step is for A and B to conclude channel χ, leaving the channel in the conclude state. Assuming $a' + b' = c$:

$$[\chi \mapsto (A : a', B : b')]_{A,B} \tag{7}$$

3. The participants then update the ledger channel to include the result of channel χ.

$$[L \mapsto (A : a + a', B : b + b')]_{A,B} \tag{8}$$

4. Now the sub-channel χ has been defunded, it can be safely discarded.

5.4.3 Topping Up a Ledger Channel

Here we show how a participant can increase their funds held in a ledger channel by depositing into it. They can do this without disturbing any sub-channels supported by the ledger channel.

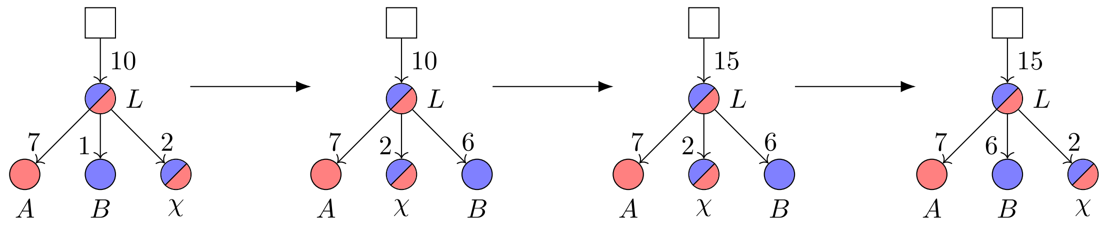


Figure 19: *Topping up a ledger channel.*

1. In this process A wants to deposit an additional a' coins into the ledger channel L. We start in the state where L contains balances for A and B, as well as funding a sub-channel, χ. With $x = a + b + c$:

$$[\![L : x]\!],\ [L \mapsto (A : a, B : b, \chi : c)]_{A,B} \tag{9}$$

2. To prepare for the deposit the participants update the state to move A's entry to the end, simultaneously increasing A's total. This is a safe operation due to the precedence rules: as the channel is currently underfunded A would still only receive a if the outcome went to chain.

$$[L \mapsto (B : b, \chi : c, A : a + a')]_{A,B} \tag{10}$$

3. It is now safe for A to deposit into the channel on-chain:

$$D_L(a')[\![L : x]\!] = [\![L : x + a']\!] \tag{11}$$

4. Finally, if required, the participants can reorder the state again:

$$[L \mapsto (A : a + a', B : b, \chi : c)]_{A,B} \tag{12}$$

5.4.4 Partial Withdrawal from a Ledger Channel

A partial checkout is the opposite of a top up: one participant has excess funds in the ledger channel that they wish to withdraw on-chain. The participants want to do this without disturbing any sub-channels supported by the ledger channels.

1. We start with a ledger channel, L, that A wants to withdraw a' coins from:

$$[\![L : x]\!],\ [L \mapsto (A : a + a', B : b, \chi : c)]_{A,B} \tag{13}$$

2. The participants start by preparing a new ledger channel, L', whose state reflects the situation they want to be in after A has withdrawn their coins. This is safe to do as this channel is currently unfunded.

$$[L' \mapsto (A : a, B : b, \chi : c)]_{A,B} \tag{14}$$

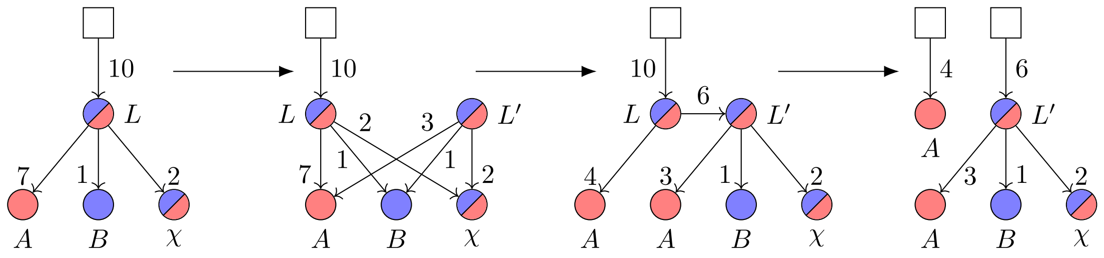


Figure 20: *Partial withdrawal from a ledger channel.*

3. They then update L to fund L' alongside the coins that A wants to withdraw. They conclude the channel in this state:

$$[L \mapsto (L' : a + b + c, A : a')]_{A,B} \tag{15}$$

4. They then finalize the outcome of L on-chain. This can be done without waiting the timeout, assuming they both signed the conclusion proof in the previous step:

$$[\![L : x \mapsto (L' : a + b + c, A : a')]\!] \tag{16}$$

5. A can then call the transfer operation to get their coins under their control.

$$\mathcal{T}(L, A, a')[\![L : x \mapsto (L' : a + b + c, A : a')]\!] = \\ [\![L : x - a' \mapsto (L' : a + b + c),\ A : a']\!] \tag{17}$$

6. At any point in the future the remaining coins can be transferred to L':

$$\mathcal{T}(L, L', a + b + c)[\![L : x \mapsto (L : a + b + c),\ A : a']\!] = \\ [\![L' : a + b + c,\ A : a']\!] \tag{18}$$

Note that A was able to withdraw their funds instantly, without having to wait for the channel timeout.

6 Nitro Protocol

Nitro protocol is an extension to Turbo protocol. In Nitro protocol, the outcome of a channel can be either an allocation or a **guarantee**. There are two on-chain redistribution operations: the transfer and the **claim**.

Nitro enables true state channel networks by allowing virtual channels, where channels are routed through intermediaries. In particular, the extra features of Nitro allow virtual channels to be safely opened and closed off-chain while maintaining the property that channels update independently[5].

6.1 Guarantees and Claims

In the section on Turbo, we introduced allocations which defined a set of destinations and totals owed to them, along with a priority order in which the destinations should receive their funds. In Nitro, we allow for these two concerns to be split between different types of channels, with one type of channel providing the destinations and totals and another type of channel specifying the priority order. This opens up the possibility of having more than one priority order for the same allocation.

The new type of channels are called **guarantee channels** and their outcomes are known as **guarantees**. A guarantee **targets** an allocation, while specifying a different priority order to pay out to the destinations. The funds held in a guarantee channel can only be paid out once the outcome of the target channel is finalized.

Adjudicator

Address	Balance	Outcome
G	3	$(L\|B)$
L		A: 2, B: 3

$$[\![G : 3 \mapsto (L|B),\ L \mapsto (A:2, B:3)]\!]$$

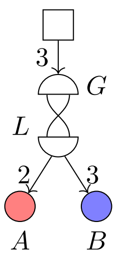


Figure 21: *Guarantee notation. The guarantee outcome of G is written $(L|B)$, where L is the allocation it targets and B is an address to be prioritized first. When targeted by a guarantee we draw the allocation, L, as a lower semicircle. The guarantee, G, is drawn as an upper semicircle. The lines between G and L depict the reprioritization, showing that the highest priority position of G (on the bottom left of the semicircle) will pay out to B.*

The format for specifying the priority order of a guarantee channels is complicated by the fact that we may not know the precise outcome or even the precise set of destinations at the time the guarantee is created. Because of this we need to enable the guarantee to apply a range of outcomes. To do this, the guarantee provides a list of addresses, with the rule that these addresses will be moved to the top of the priority order if they appear in the outcome. For example, if we have the guarantee $(L|A, B, C)$ and the outcome of L is $(C:1, D:2, A:4)$, then the guarantee will pay out as though the outcome was $(A:4, C:1, D:2)$.

[5]It is possible to implement virtual channels in Turbo but only by breaking the constraint that channels update independently. We believe this constraint will be important when running large scale state channel networks.

Extending the terminology for allocations, we say that a guarantee ‘G **can afford** x for B’, if B would receive at least x coins, were the coins currently held by G to be paid out according to G’s prioritization of its target.

Adjudicator

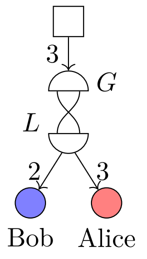
Address	Balance	Outcome
G	3	$(L\|\text{Alice})$
L		Bob: 2, Alice: 3

$$\xrightarrow{\mathcal{C}(G, \text{Alice}, 3)}$$

Adjudicator

Address	Balance	Outcome
G		$(L\|\text{Alice})$
L		Bob: 2
Alice	3	

$$[\![G : 3 \mapsto (L|\text{Alice}),\ L \mapsto (\text{Bob} : 2, \text{Alice} : 3)]\!]$$



$$[\![G \mapsto (L|\text{Alice}),\ L \mapsto (\text{Bob} : 2),\ \text{Alice} : 3]\!]$$

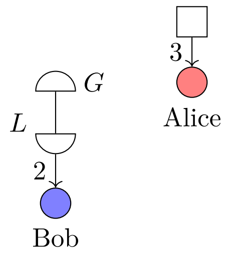


Figure 22: *The claim operation is used to move funds out of a guarantee. Note that in diagram notation, we update the reprioritization lines to reflect the new outcome: after the claim, L only allocates to Bob, so there is only one line from G to L.*

Nitro adds the **claim** operation, $\mathcal{C}(G, A, x)$, to the existing transfer, deposit and withdraw operations. If G acts as guarantee for L and can afford x for A, then $\mathcal{C}(G, A, x)$ has the following three effects:

- Reduces the funds held in channel G by x.
- Increases the funds held in channel A by x.
- Reduces the amount owed to A in the outcome of L by x.

Otherwise, the claim operation has no effect.

6.2 Redistributing

Reasoning about redistribution in Nitro is more complicated than in Turbo. For a start, it is possible to construct situations where the same outcome can lead to different values, depending on the order in which guarantees are claimed. Figure 23 shows one of these situations.

Despite this issue, it is still possible to make some statements about redistribution in Nitro, in particular putting some lower bounds on how funds are distributed. For example, in the example in figure 23 one thing we can definitely

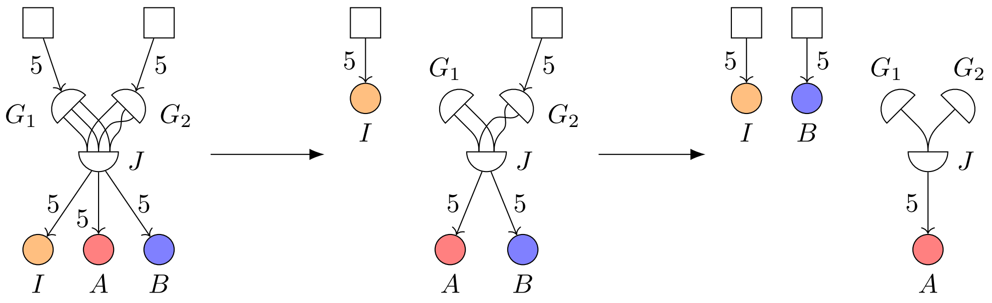


Figure 23: *Guarantee claim ordering problem. In the diagram both G_1 and G_2 guarantee J's outcome with I as first priority but with different second priorities. If G_1 is claimed first, then when G_2 is claimed the funds go to B. If G_2 is claimed first (not shown), then when G_1 is claimed the funds go to A. Whether A or B ultimately gets paid depends on the order that the guarantees are claimed.*

say is that participant I will receive their 5 coins, even though we cannot say anything about how the remaining 5 coins will be distributed between A and B. These lower bounds prove to be enough to handle the constructions used in the rest of the chapter.

Nitro Lower-bound Value Algorithm

We can calculate this lower bound with a modification the Turbo Value Algorithm:

1. Choose a topological ordering, `OrderedNodes`, for the network.
2. Create a mapping, `Values`, from `OrderedNodes` to $\mathbb{Z}^+$. Initialize this mapping by setting `Values`$[n]$ to be the balance held for n in the adjudicator, for each $n \in$ `OrderedNodes` (with `Values`$[n] = 0$ if n's address does not appear).
3. For each $n \in$ `OrderedNodes`$[n]$ (taken in order):
    (a) If n is a guarantee channel, do nothing.
    (b) Otherwise, n is an allocation channel:
        i. Run the Minimum Payout Calculation on n and the guarantees that target it, to calculate the minimum payouts, `MinPayouts`, for each of destination channels.
        ii. For each destination, d, increase `Value`$[d]$ by `MinPayouts`$[d]$.
4. Then `Values`$[n]$ gives the value of node n.

Note that by taking the minimum independently on different allocation/guarantee groupings, we end up with an algorithm that strictly underestimates the actual value in some cases. This is not the case for any of the constructions presented in this paper.

Minimum Payout Calculation

In this calculation we will consider an allocation channel, L, whose outcome allocates $a_1 \dots a_m$ to destination addresses $D_1 \dots D_m$, and a set of guarantees $G_1 \dots G_n$ which target L. We want to calculate the minimum payout, p_i, that each destination will receive when all possible payout orders are considered.

Each guarantee, G_i, induces a permutation π_i on the destination addresses, so that G_i prioritizes the outcomes in the order $D_{\pi_i(1)} \dots D_{\pi_i(m)}$.

We start in a state where the values of channel L and the guarantees are known, with G_i having value $v_i = \texttt{Value}[G_i]$. We will assume the value of L itself is 0. We are free to do this because, if $\texttt{Value}[L] = x > 0$, then for the purpose of running the algorithm we can write the problem in an equivalent way, by adding a guarantee G_{n+1} that has value x, that targets L and that has $\pi_{n+1}(k) = k$.

If $\sum v_i > \sum a_j$, then we say the system is overfunded. In this case, we know that all destinations will receive their allocations, so $p_i = a_i$ regardless of the order of payout. Otherwise, we let $p_{ij} > 0$ be the amount paid out from guarantee G_i to destination D_j and introduce the following set of constraints to ensure that we only consider situations where no funds are left in the guarantees:

$$\sum_j p_{ij} = v_i \tag{19}$$

It is useful to introduce the deficit, $\delta_j > 0$, for the destination D_j, defined by the equations:

$$\delta_j + \sum_i p_{ij} = a_j \tag{20}$$

Finally we can write down the set of constraints that encode the priority order of the guarantees:

$$\begin{aligned} p_{i\pi_i(2)} > 0 &\Rightarrow \delta_{\pi_i(1)} = 0 \\ p_{i\pi_i(3)} > 0 &\Rightarrow \delta_{\pi_i(2)} = \delta_{\pi_i(1)} = 0 \\ &\vdots \\ p_{i\pi_i(m)} > 0 &\Rightarrow \delta_{\pi_i(m-1)} = \cdots = \delta_{\pi_i(1)} = 0 \end{aligned} \tag{21}$$

Note that we can rewrite these constraints in product form, e.g. $p_{i\pi_i(3)}(\delta_{\pi_i(2)} + \delta_{\pi_i(1)}) = 0$, making it clear that they are non-linear.

We can then calculate $p_i = a_i - \delta_i^*$, where δ_i^* is found by minimising δ_i subject to these constraints.

In general, calculating the minimum payout therefore involves solving a constrained optimization problem, with non-linear constraints. In practice, for all the calculations required for the constructions in this paper, it is sufficient to look at two special cases: (i) when the allocation is fully funded and (ii) when there is only a single guarantee. In the fully funded case, where $\sum v_i = \sum a_i$, it is easy to see that $p_i = a_i$, just as in the overfunded case. In the single guarantee case, the payouts are fully determined, so it is easy to calculate the minimum payout.

6.3 Virtual Channels

A virtual channel is a channel between two participants who do not have a shared on-chain deposit, supported through an intermediary. We will now give the construction for the simplest possible virtual channel, between A and B through a shared intermediary, I. Our starting point for this channel is a pair of ledger channels, L and L', with participants $\{A, I\}$ and $\{B, I\}$ respectively.

$$\llbracket L : x, L' : x \rrbracket,\ [L \mapsto (A : a, I : b)]_{A,I},\ [L' \mapsto (B : b, I : a)]_{B,I} \tag{22}$$

where $x = a + b$. The participants want to use the existing deposits and ledger channels to fund a virtual channel, χ, with x coins.

In order to do this the participants will need three additional channels: a joint allocation channel, J, with participants $\{A, B, I\}$ and two guarantor channels G and G' which target J. The setup is shown in figure 24.

$$\begin{tikzcd}
& \llbracket L : x, L' : x \rrbracket \arrow[dl, dash] \arrow[dr, dash] & \\
{[L \mapsto (G : x)]_{A,I}} \arrow[d, dash] & & {[L' \mapsto (G' : x)]_{B,I}} \arrow[d, dash] \\
{[G \mapsto (J|\chi, A, I)]_{A,I}} \arrow[dr, dash] & & {[G' \mapsto (J|\chi, B, I)]_{B,I}} \arrow[dl, dash] \\
& {[J \mapsto (\chi : x, I : x)]_{A,B,I}} &
\end{tikzcd}$$

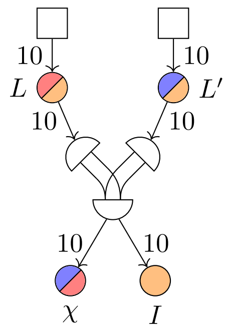


Figure 24: *Virtual channel construction.*

We will cover the steps for safely setting up this construction in section 6.5. In the next section, we will explain why this construction can be considered to fund the channel χ.

6.4 Offloading Virtual Channels

Similarly to the method for ledger channel construction, we will show that the virtual channel construction funds $\mathcal{X}$ by demonstrating how any one of the participants can offload the channel $\mathcal{X}$, thereby converting it to an on-chain channel that holds its own funds.

We will first consider the case where A wishes to offload $\mathcal{X}$. A proceeds as follows:

1. A starts by finalizing all their finalizable outcomes on-chain:

$$[\![L : x \mapsto (G : x),\ L' : x, G \mapsto (J|\mathcal{X}, A, I),\ J \mapsto (\mathcal{X} : x, I : x)]\!] \tag{23}$$

   Although A has the power to finalize L, G and J, they are not able to finalize L'. Thankfully, this does not prevent them from offloading $\mathcal{X}$.

2. A then calls $\mathcal{T}(L, G, x)$ to move the funds from L to G:

$$[\![L' : x, G : x \mapsto (J|\mathcal{X}, A, I),\ J \mapsto (\mathcal{X} : x, I : x)]\!] \tag{24}$$

3. Finally A calls $\mathcal{C}(G, \mathcal{X}, x)$ to move the funds from G to $\mathcal{X}$.

$$[\![L' : x, G \mapsto (J|\mathcal{X}, A, I),\ J \mapsto (I : x),\ \mathcal{X} : x]\!] \tag{25}$$

As G has $\mathcal{X}$ as top priority, the operation is successful.

By symmetry, the previous case also covers the case where B wants to offload. The final case to consider is the one where I wants to offload the channel and reclaim their funds. This is important to ensure that A and B cannot lock I's funds indefinitely in the channel.

1. I starts by finalizing all their finalizable outcomes on-chain:

$$\begin{multline}[\![L : x \mapsto (G : x),\ L' : x \mapsto (G' : x), G \mapsto (J|\mathcal{X}, A, I),\\ G' \mapsto (J|\mathcal{X}, B, I),\ J \mapsto (\mathcal{X} : x, I : x)]\!]\end{multline} \tag{26}$$

2. I then transfers funds from the ledger channels to the guarantee channels by calling $\mathcal{T}(L, G, x)$ and $\mathcal{T}(L', G', x)$:

$$[\![G : x \mapsto (J|\mathcal{X}, A, I),\ G' : x \mapsto (J|\mathcal{X}, B, I),\ J \mapsto (\mathcal{X} : x, I : x)]\!] \tag{27}$$

3. Then I claims on one of the guarantees, e.g. $\mathcal{C}(G, \mathcal{X}, x)$ to offload $\mathcal{X}$:

$$[\![G \mapsto (J|\mathcal{X}, A, I),\ G' : x \mapsto (J|\mathcal{X}, B, I),\ J \mapsto (I : x),\ \mathcal{X} : x]\!] \tag{28}$$

4. After which, I can recover their funds by claiming on the other guarantee, $\mathcal{C}(G', I, x)$:

$$[\![G \mapsto (J|\mathcal{X}, A, I),\ G' \mapsto (J|\mathcal{X}, B, I),\ \mathcal{X} : x,\ I : x]\!] \tag{29}$$

Note that I has to claim on both guarantees, offloading χ before being able to reclaim their funds. The virtual channel became a direct channel and the intermediary was able to recover their collateral.

6.5 Examples

In this section we present a sequence of network states written in terms of universally finalizable outcomes, where each state differs from the previous state only in one channel.

6.5.1 Opening a Virtual Channel

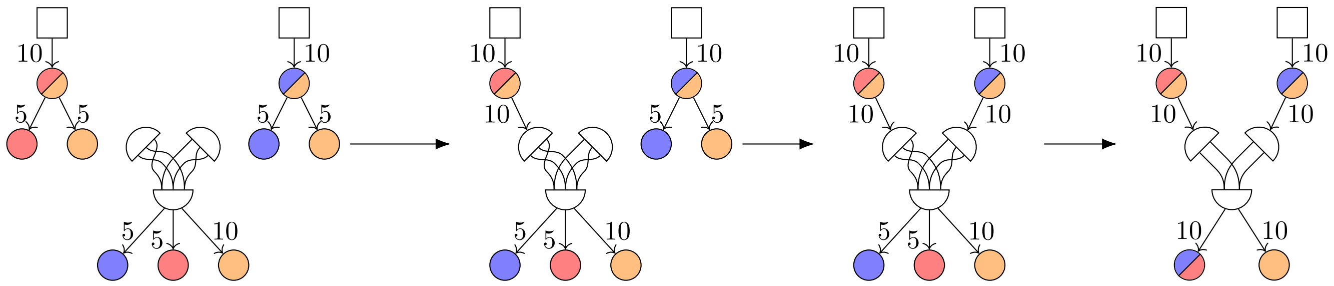


Figure 25: *Opening a virtual channel*

The procedure for opening a virtual channel is as follows:

1. Start in the state given in equation (22):

$$[\![L : x, L' : x]\!] \tag{30}$$
$$[L \mapsto (A : a, I : b)]_{A,I} \tag{31}$$
$$[L' \mapsto (B : b, I : a)]_{B,I} \tag{32}$$

2. A and B bring their channel χ to the funding point:

$$[\chi \mapsto (A : a, B : b)]_{A,B} \tag{33}$$

3. In any order, A, B and I setup the virtual channel construction:

$$[J \mapsto (A : a, B : b, I : x)]_{A,B,I} \tag{34}$$
$$[G \mapsto (J|\chi, A, I)]_{A,I} \tag{35}$$
$$[G' \mapsto (J|\chi, B, I)]_{B,I} \tag{36}$$

4. In either order switch the ledger channels over to fund the guarantees:

$$[L \mapsto (G : x)]_{A,I} \tag{37}$$
$$[L' \mapsto (G' : x)]_{B,I} \tag{38}$$

5. Switch J over to fund χ:

$$[J \mapsto (\chi : x, I : x)]_{A,B,I} \tag{39}$$

We give a visual representation of this procedure in figure 25.

6.5.2 Closing a Virtual Channel

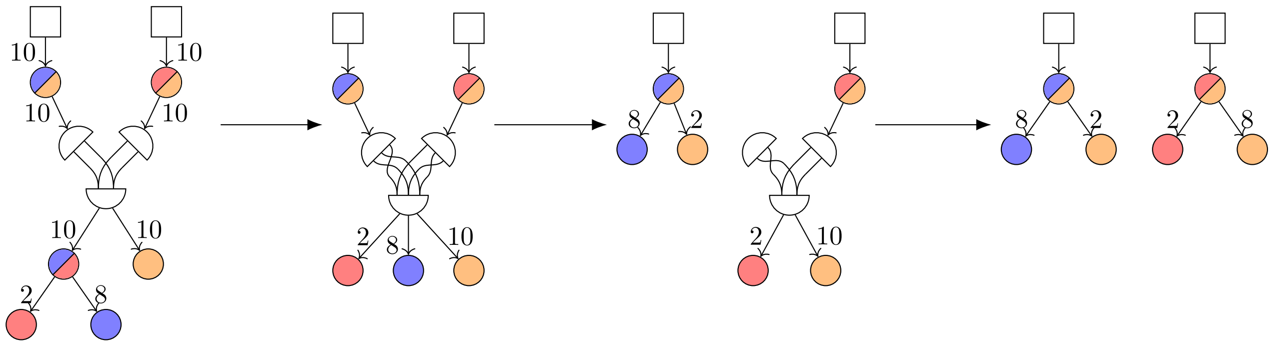


Figure 26: *Closing a virtual channel*

The same sequence of states, when taken in reverse, can be used to close a virtual channel:

1. Participants A and B finalize χ by signing a conclusion proof:

$$[\chi \mapsto (A : a', B : b')]_{A,B} \tag{40}$$

2. A and B sign an update to J to take account of the outcome of χ. I will accept this update, provided that their allocation of x coins remains the same:

$$[J \mapsto (A : a', B : b', I : x)]_{A,B,I} \tag{41}$$

3. In either order switch the ledger channels to absorb the outcome of J, defunding the guarantor channels in the process:

$$[L \mapsto (A : a', I : b')]_{A,I} \tag{42}$$
$$[L' \mapsto (B : b', I : a')]_{B,I} \tag{43}$$

4. The channels χ, J, G and G' are now all defunded, so can be discarded.

It is also possible to do top-ups and partial checkouts from a virtual channel.

6.5.3 Virtual Channel with Three Participants

So far we have primarily focussed on channels with two participants. The techniques here all generalise to n-participant channels. In figure 27, we give an example construction for a virtual channel between three participants. The opening and closing of this channel follows the same pattern as the two participant case.

$$\begin{tikzcd}
 & \llbracket L : x, L' : x, L'' : x \rrbracket \arrow[dl, -] \arrow[d, -] \arrow[dr, -] & \\
{[L \mapsto (G : x)]_{A,I}} \arrow[d, -] & {[L' \mapsto (G' : x)]_{B,I}} \arrow[d, -] & {[L'' \mapsto (G'' : x)]_{C,I}} \arrow[d, -] \\
{[G \mapsto (J|\chi, A, I)]_{A,I}} \arrow[dr, -] & {[G' \mapsto (J|\chi, B, I)]_{B,I}} \arrow[d, -] & {[G'' \mapsto (J|\chi, B, I)]_{C,I}} \arrow[dl, -] \\
 & {[J \mapsto (\chi : x, I : x)]_{A,B,C,I}} \arrow[d, -] & \\
 & {[\chi \mapsto (A : a, B : b, C : c)]_{A,B,C}} &
\end{tikzcd}$$

Figure 27: *Virtual channel with three participants. Here $x = a + b + c$.*

7 Conclusion

In this paper, we have presented a protocol to allow the construction of state channel networks. The channels in these networks operate independently from one another and follow the same rules of operation regardless of whether they are funded directly or via other channels. The examples we have given make it clear how to open and close channels off-chain, while also performing many operations that are important in the practical operation of these channels, such as top-ups and partial withdrawals.

By splitting the extraction of value from a state channel into finalization and redistribution, we have provided an approach for reasoning about the correctness of state channels that can be applied beyond the constructions presented in this paper. For both of the protocols we present, we have provided algorithms to calculate the value obtainable, allowing for the creation of software to automate these calculations.

Acknowledgments

I would like to thank James Prestwich, Andrew Stewart and Chris Buckland for multiple discusssions which helped shape the work in this paper. George Knee, Kristina Hostakova and Lisa Eckley all provided important feedback on the draft, resulting in multiple improvements to the paper. This work is funded through an Ethereum Foundation Grant.

Post publication edits: I would like to thank Alexander Poddey for finding and correcting notation errors in the Minimum Payout Calculation and in the Partial Withdrawal from a Ledger Channel.