Nitro Protocol

Tom Close

February 5, 2019

Abstract

State channels are an important technique for scaling blockchains, allowing a fixed set of participants to trustlessly execute a series of state transitions off-chain, in order to determine how a set of assets should be distributed between them. In this paper we present constructions that allow state channels to fund one another, significantly reducing the number of on-chain deposits required. Unlike in previous constructions, channels funded by parent channels operate identically to channels funded on-chain, making it easier to reason about the applications that run within them. We develop the logic to prove the correctness of our constructions.

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.

Unlike many other scaling techniques, state channels provide a way to run arbitrary state update protocols, instead of just providing a method for realizing transfers off-chain.

Beyond scaling, state channels bring instant finality to blockchain transactions: value can be considered to be transferred at the moment when a state channel update is received. The holder of a fully signed state does not need to wait for the transaction to be mined, safe in the knowledge that they have the right to claim the assets on-chain at a future point of time of their choosing.

In their naive form, each state channel needs to have a corresponding *state deposit* - a set of assets held in escrow on-chain, to be distributed according to the outcome of the channel. Each time a state channel is opened, 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.

1.1 Ledger Channels and Virtual Channels

- graph of direct channels with multiple lines between nodes

- graph of ledger channels with single lines between nodes

- graph of virtual channels with hub + spoke

- image of virtual channels, talking about the agreements

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 channel 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 timelocked 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. They precisely 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.

2.1 Our contribution

This paper specifies in detail how to build virtual channels on top of the Force-Move state channel framework. Along with the construction we provide some high-level proofs to justify that the construction is correct (TODO: need to add this to the appendix).

Unlike Perun's detailed construction, our state channel networks place no restrictions on the operation of virtual channels: we remove the channel validity time, instead allowing intermediaries to 'off-load' their channels as protection against arbitrarily long locking of their deposit. The off-load procedure converts a virtual channel to a regular on-chain channel, without interrupting the operation within that channel.

The protocol described here readily extends to virtual channels between n-parties, although we do not give this construction in the paper.

We will now proceed with a more technical exposition of our protocol, which achieves the functionality outlined above.

3 Modelling State Channel Networks

In this section we will present a simple model for state channel networks, which will serve as a foundation for the protocols introduced later in the paper. The model is opinionated, making choices about how state channel networks should behave, but also captures the essential features seen in other approaches.

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 real-world assets.

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 arbitrary address, A, we assume that the properties of the signature scheme and hashing algorithm make it neither possible to find a private key for A nor to find a channel whose address is A.

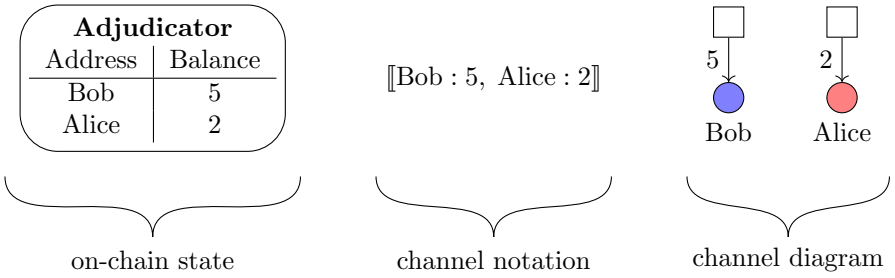


Figure 1: *Three different ways of representing the situation where Bob has* 5 *coins stored against his address in the adjudicator and Alice has* 2*.*

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 private key. In practice the withdrawal should also specify the blockchain address where the funds should be sent. A potential method signature is

[1] In practice, the adjudicator functionality could be split across multiple smart contracts.

`withdraw(fromAddr, toAddr, amount, signature)`, where `signature` is A's signature of the other parameters [2].

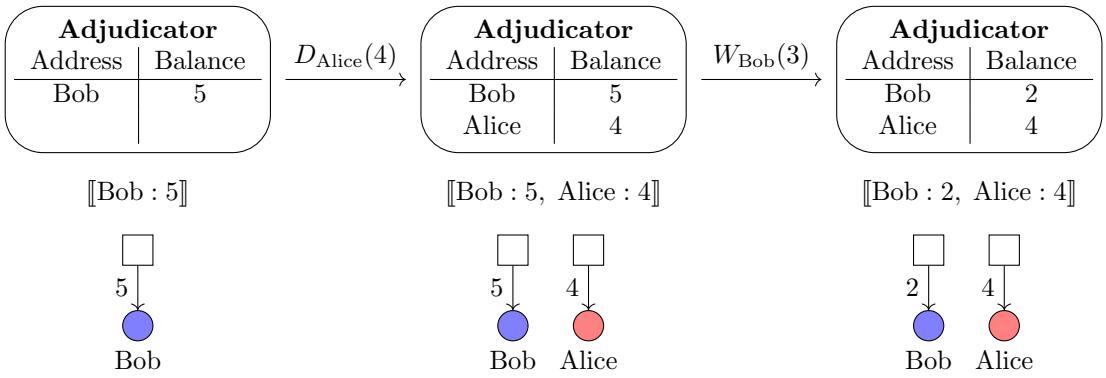


Figure 2: *Deposits and withdrawals.*

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 can be increased and decreased through deposits and withdrawals. Anyone can deposit into an address of either type[3] but withdrawals can only occur 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, we think of a state channel as a off-chain protocol followed by a set of participants, enabling them to reach an **outcome** that can be used to change the balances on the chain. In general, the outcome can be an arbitrary data structure, the format and interpretation of which is specified by the state channel network protocol being used. For example, the *allocation* is a simple type of outcome used by both Turbo and Nitro protocols. The data in an allocation represents a list of addresses paired with totals, specifying how the channel's balance should be distributed.

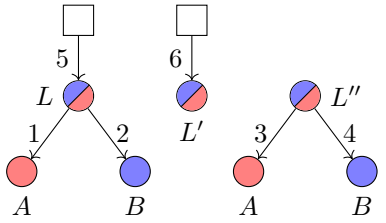
We separate the process of updating the balances according to the channel outcome into two stages: finalization and redistribution. **Finalization** is the process of getting the outcome of the state channel into the adjudicator on chain, where it will be stored against the channel address. **Redistribution** is the process of updating the balances in the adjudicator, according to the outcome.

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

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

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. In colouring the channels, we have assumed the participants in each channel are A and B.*

How an outcome is finalized is a question of the rules of operation of a state channel. The rules of operation are somewhat orthogonal to the network protocols that are the focus of this paper. The ForceMove protocol is an example of a set of state channel operation rules. The protocols presented here are known to work with ForceMove channels but could also likely be made to work with other operation rules.

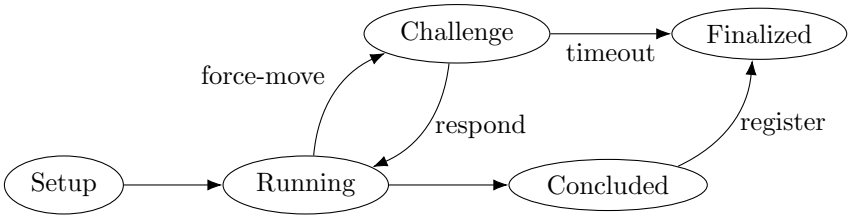


Figure 4: *ForceMove Channel Operation. The parts of the adjudicator responsible for storing challenges are omitted from the diagram.*

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 by providing a sequence of signed statements to the force-move operation; this starts a timeout period; if no other participant responds during 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, which contains the channel outcome to produce an object 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, a crucial property is 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.

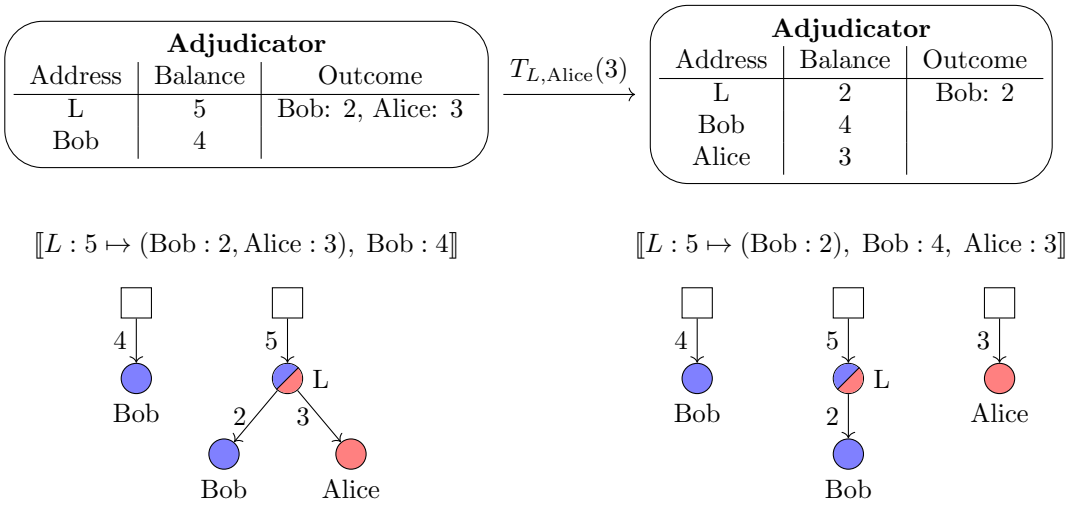


Figure 5: *A example of a redistribution operation from Turbo protocol. Here the transfer operation is used to move* 3 *coins out of channel* L *to Alice.*

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.

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 discuss how to reason about state channels.

4 Reasoning about State Channels

- paper aims to put in place a framework for reasoning about the correctness of state channel constructions - reasoning about what I hold off-chain and what if means to me

- how to extract value - states —¿ outcomes —¿ money - reducing channel diagrams –¿ value

- outcome diagrams - fundamental rule of state channels - if two states are worth the same, I will transition between them - the "simple rule of state channels" - we ignore other factors - value - funded ? - offloaded

- safe - this is the guarantee that if a participant stops at any point other participants don't lose out

- allows rewriting - [diagram] example: closing off-chain

- two questions (finalization + redistribution) - what can I definitely finalize? - what can I definitely redistribute to myself?

- protocol design: how can I move between states in single moves that keep the value the same - .. when we don't allow atomic changes across channels

- presenting a construction / protocol - in particular when presenting a protocol we must demonstrate a series single state updates, demonstrate that the value is preserved - in particular the way we do this: - demostrate a construction funds a channel - demonstrate we can build it from another state - give a series of waypoint states - universally finalizable outcomes - of a special type of channel - consensus channels, running a particilar protocol - use the specifics of this channel to say we can move between them in a safe manner

4.1 Finalizable outcomes

- definition: statement - definition: channel state - definition: adjudicator state - definition: system state

- the rules of strategies

subsection consensus game [diagram] - pictorial representation of consensus game

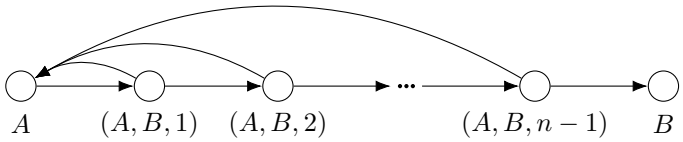


Figure 6: *Cool, huh?*

5 Turbo Protocol

The outcome of a Turbo channel is always an **allocation**. 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. If the outcome of channel A includes the pair $B{:}x$, we say that 'A **owes** x to B'. 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.

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

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 $T_{A,B}(x)$ fails, leaving the on-chain state unchanged.

Example 5.1. In the following example, we have a channel, L, which holds 10 coins and has an outcome, $(A : 3, B : 2, \chi : 5)$, which has been finalized on-chain. As L can afford 5 for χ the following transfer operation is successful:

$$T_{L,\chi}(5)[\![L : 10 \mapsto (A : 3, B : 2, \chi : 5)]\!] = [\![L : 5 \mapsto (A : 3, B : 2), \chi : 5]\!] \qquad (1)$$

We give a python implementation of the Turbo adjudicator in the appendix.

5.1 Ledger Channels

A **ledger** channel is a channel which uses its own funding to fund other channels sharing the same set of participants. By doing this, a ledger channel allows these **sub-channels** to be opened, funded and closed without any on-chain operations.

To see how this works, consider the following setup where a ledger channel, L, allocates the funds it holds to participants A and B and channel χ:

$$[\![L : 10]\!], [L \mapsto (A : 2, B : 3, \chi : 5)]_{A,B} \tag{2}$$

We have chosen the simplest example here, where L is funded by the coins it holds on-chain, but it is completely possible to have ledger channels themselves funded by other ledger channels or (later) by virtual channels.

We claim that the ledger channel L funds the channel χ in the above setup. This is because either participant has the power to convert the situation above into a situation where χ is funded on-chain:

$$T_{L,\chi}(5)[\![L : 10 \mapsto (A : 2, B : 3, \chi : 5)]\!] = [\![L : 10 \mapsto (A : 2, B : 3), \chi : 5]\!] \tag{3}$$

After this operation, we say that the channel χ has been **offloaded**. Note that we do not need to wait for χ to complete before offloading it. The offload converts χ from an off-chain sub-channel to an on-chain direct channel, without interrupting its operation in any way.

The offload should be seen as an action of last-resort. It is important that offloading is allowed so that either player can realize the value in channel χ if required, but it has the downside of forcing all sub-channels supported by L to be closed on-chain. It is in the interest of both participants to open and close sub-channels collaboratively. We next show how this can be accomplished safely.

Under the Turbo protocol, all ledger channels are regular ForceMove channels running the Consensus Game.

5.1.1 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:

$$[\![L : x]\!],\ [L \mapsto (A : a, B : b)]_{A,B} \tag{4}$$

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

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

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

$$[L \mapsto (A : a - a', B : b - b', \chi : a' + b')]_{A,B} \tag{6}$$

5.1.2 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 $x = a + b + c$:

$$[\![L : x]\!],\ [L \mapsto (A : a, B : b, \chi : c)]_{A,B} \tag{7}$$

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

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

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

$$[L \mapsto (A : a + a', B : b + b')]_{A,B} \tag{9}$$

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

5.1.3 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.

1. In this process A wants to deposit an additional a' coins into the 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 $x = a + b + c$:

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

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.

$$[L \mapsto (B : b, \chi : c, A : a + a')]_{A,B} \tag{11}$$

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

$$D_L(a')[\![L : x]\!] = [\![L : x + a']\!] \tag{12}$$

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

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

5.1.4 Partial checkout from a ledger channel

Figure 7: *Cool, huh?*

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:

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

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.

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

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

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

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:

$$[\![L : x \mapsto (L' : a + b + c, A : a')]\!] \tag{17}$$

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

$$T_{L,A}(a')[\![L : x \mapsto (L : a + b + c, A : a')]\!] = \\ [\![L : x - a' \mapsto (L' : a + b + c),\ A : a]\!] \tag{18}$$

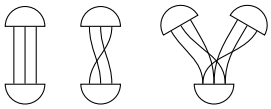
6. At any point in the future the remaining coins can be transferred to L':

$$T_{L,L'}(a+b+c)[\![L : x \mapsto (L : a+b+c),\ A : a']\!] = \\ [\![L' : a+b+c,\ A : a']\!] \quad (19)$$

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**. A guarantee outcome specifies the address of a target allocation channel; the protocol specifies how this guarantee may be used to pay debt on its behalf. When paying debt, a guarantee can be used to alter the payout priority of the allocation outcome of its target address.



We will use the notation $(L|A_1, A_2, \ldots, A_m)$ for a guarantee with target L, which prioritizes payouts to A_1 above A_2, A_2 above A_3, and so on. Any addresses which occur in the outcome of L but not in the guarantee are prioritized after A_m, in the order they occur in the outcome. We say a guarantor channel, G, which targets an allocation channel, L, 'can afford x for A', if A would receive at least x coins, were the coins currently held in A to be paid out according to G's reprioritization of L's allocation.

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

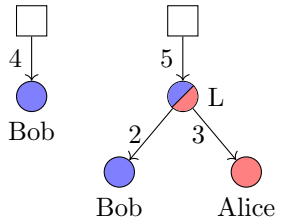
- Reduces the funds held in channel G by x.
- Increases the funds held in channel A by x.
- Reduces the amount owed to A in the outcome of L by x.

Otherwise, the claim operation has no effect.

Adjudicator

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

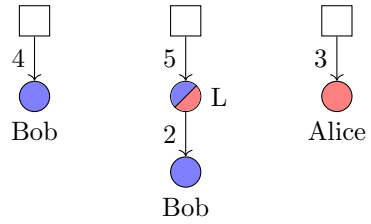
$[\![G : 3 \mapsto (L|\text{Alice}),\ L \mapsto (\text{Bob} : 2, \text{Alice} : 3)]\!]$



Adjudicator

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

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



Example 6.1. In the following example, we have a guarantor channel, G, which holds 5 coins and guarantees L's allocation, with B as highest priority.

$$C_{G,B}(5)[\![G : 5 \mapsto (L|B), L \mapsto (A : 5, B : 5)]\!] = [\![G \mapsto (L|B), L \mapsto (A : 5), B : 5]\!] \quad (20)$$

Note that after the claim has gone through, L's debt to B has decreased.

We give a python implementation of an adjudicator implementing the Nitro protocol in the appendix.

6.1 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.

$$[\![L : x, L' : x]\!],\ [L \mapsto (A : a, I : b)]_{A,I},\ [L' \mapsto (B : b, I : a)]_{B,I} \quad (21)$$

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 8.

We will cover the steps for safely setting up this construction in section 6.3. In the next section, we will explain why this construction can be considered to fund the channel χ.

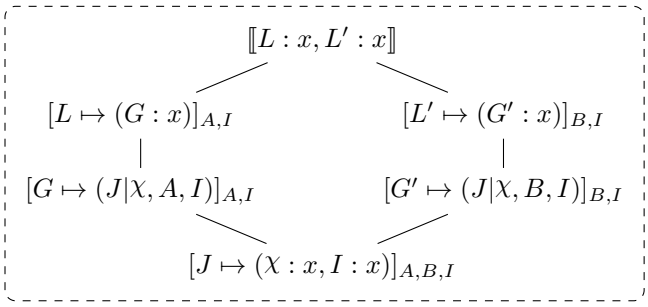


Figure 8: *Virtual channel construction*

6.2 Offloading Virtual Channels

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

We will first consider the case where A wishes to offload χ. A proceeds as follows:

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

$$[\![L : x \mapsto (G : x),\ L' : x, G \mapsto (J|\chi, A, I),\ J \mapsto (\chi : x, I : x)]\!] \tag{22}$$

   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 χ.

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

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

3. Finally A calls $C_{G,A}(\chi)$ to move the funds from G to χ.

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

As G has χ 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:

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

2. I then transfers funds from the ledger channels to the virtual channels by calling $T_{L,G}(x)$ and $T_{L',G'}(x)$:

$$[\![G : x \mapsto (J|\chi, A, I),\ G' : x \mapsto (J|\chi, B, I),\ J \mapsto (\chi : x, I : x)]\!] \tag{26}$$

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

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

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

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

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.

6.3 Opening and Closing Virtual Channels

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. We claim that this sequence of states can be used to derive a safe procedure for opening a virtual channel, where the value of the network remains unchanged throughout for all participants involved. We justify this claim in the appendix.

The procedure for opening a virtual channel is as follows:

1. Start in the state given in equation (21):

$$[\![L : x, L' : x]\!] \tag{29}$$
$$[L \mapsto (A : a, I : b)]_{A,I} \tag{30}$$
$$[L' \mapsto (B : b, I : a)]_{B,I} \tag{31}$$

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

$$[\chi \mapsto (A : a, B : b)]_{A,B} \tag{32}$$

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

$$[J \mapsto (A : a, B : b, I : x)]_{A,B,I} \tag{33}$$
$$[G \mapsto (J|\chi, A, I)]_{A,I} \tag{34}$$
$$[G' \mapsto (J|\chi, B, I)]_{B,I} \tag{35}$$

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

$$[L \mapsto (G : x)]_{A,I} \tag{36}$$
$$[L' \mapsto (G' : x)]_{B,I} \tag{37}$$

5. Switch J over to fund χ:

$$[J \mapsto (\chi : x, I : x)]_{A,B,I} \tag{38}$$

We give a visual representation of this procedure in figure 9.

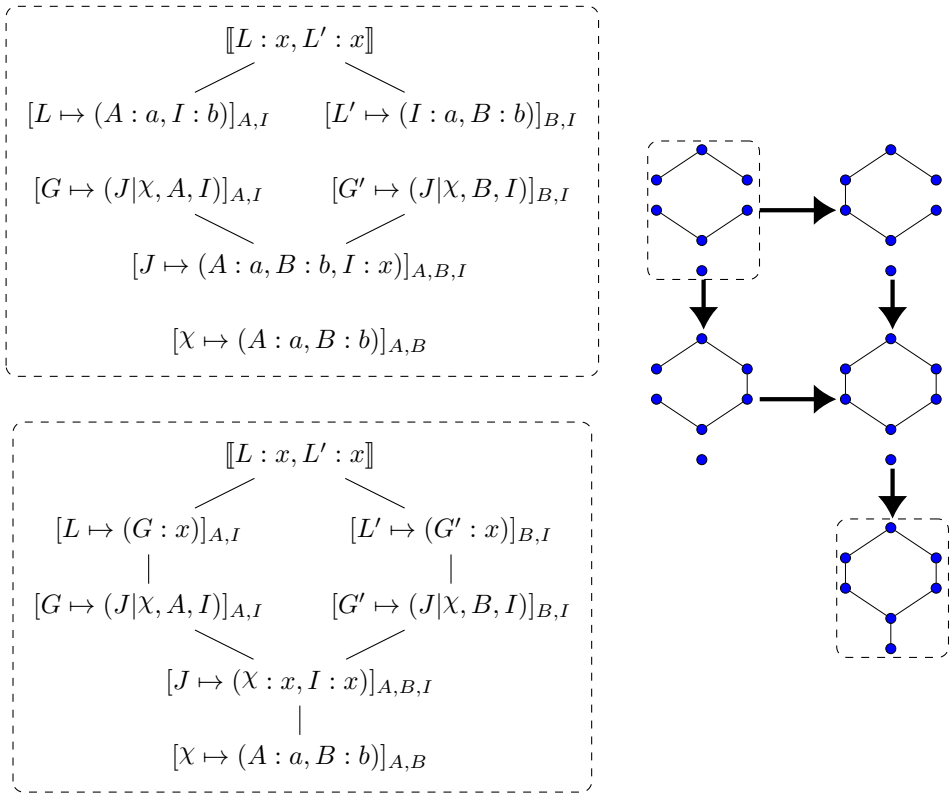
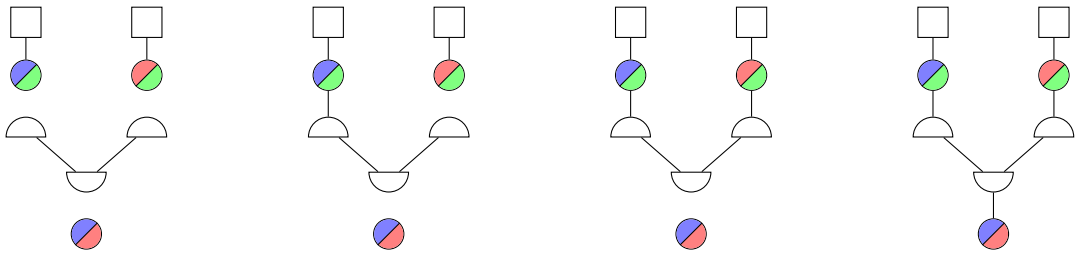


Figure 9: *Opening a virtual channel*



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:

$$[\chi \mapsto (A : a', B : b')]_{A,B} \tag{39}$$

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:

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

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

$$[L \mapsto (A : a', I : b')]_{A,I} \tag{41}$$
$$[L' \mapsto (B : b', I : a')]_{B,I} \tag{42}$$

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.

7 Acknowledgements

- Andrew Stewart - George Knee - James Prestwich - Chris Buckland - Magmo team

8 Appendix

THIS APPENDIX IS STILL A WIP.

8.1 Overview of ForceMove

The ForceMove protocol describes the message format and the supporting on-chain behaviour to enable generalized, n-party state channels on any blockchain that supports Turing-complete, general-purpose computation. Here we give a brief overview of the protocol to the level required to understand the rest of the paper. For a more comprehensive explanation please refer to [].

`participants`	`address[]`	P	The addresses used to sign updates to the channel.
`nonce`	`unit256`	k	Chosen to make the channel's address unique.
`gameLibrary`	`address`	L	The address of the gameLibrary, which defines the transition rules for this channel
`challengeDuration`	`unit256`	η	
`turnNum`	`unit256`	i	Increments as new states are produced.
`balances`	`(address, uint256)[]`	β	Current *outcome* of the channel.
`isFinal`	`bool`	f	
`data`	`bytes`	δ	
`v`	`uint8`		ECDSA signature of the above arguments by the moving participant.
`r`	`bytes32`		
`s`	`bytes32`		

Table 1: *ForceMove state format*

A ForceMove **state**, $\sigma^i_\chi(\beta, f, \delta)$, is specified by **turn number**, i, a set of **balances**, β, a boolean flag **finalized**, $f \in \{T, F\}$, and a chunk of unstructured **game data**, δ, that will be interpreted by the game library. The balances can be thought of as an ordered set of (`address`, `uint256`) pairs, which specify how any funds allocated to the channel should be distributed if the channel were to finalize in the current state.

In order for a state, σ^i_χ, to be valid it must be signed by participant, p_j, where $j = i\%n$ is the remainder mod n. This requirement specifies that participants in the channel must take turns when signing states.

The game library is responsible for defining a set of states and allowed transitions that in turn define the 'application' that will run inside the state channel. It

does this by defining a single boolean function, $t_L(i, \beta, \delta, \beta', \delta') \to \{T, F\}$. This function is used to derive an overall boolean transition function, t, specifying whether a transition between two states is permitted under the rules of the protocol:

$$\begin{aligned} t(\sigma^i_\chi(\beta, f, \delta), \sigma^j_{\chi'}(\beta', f', \delta')) \Leftrightarrow & \chi = \chi' \wedge j = i + 1 \wedge \\ & [(\neg f \wedge \neg f' \wedge j \leq 2n \wedge \beta = \beta' \wedge \delta = \delta') \vee \\ & (\neg f \wedge \neg f' \wedge j > 2n \wedge t_L(n, \beta, \delta, \beta', \delta')) \vee \\ & (f' \wedge \beta = \beta' \wedge \delta' = 0)] \end{aligned}$$

In all transitions the channel properties must remain unchanged and the turn number must increment. There are then three different modes of operation. The first mode applies in the first $2n$ states (assuming none of these are finalized) and in this mode the balances and game data must remain unchanged. As we will see later, these states exist so that the channel can be funded safely. We refer to the first n states as the **pre-fund setup** states and the subsequent n as the **post-fund setup** states. The second mode applies to the 'normal' operation of the channel, when the game library is used to determine the allowed transitions. The final mode concerns the finalization of the channel: at any point the current participant can choose to exit the channel and lock in the balances in the current state. Once this happens the only allowed transitions are to additional finalized states. Because of this, we have no further use for the game data, δ, so can remove this from the state. Once a sequence of n finalized states have been produced the channel is considered closed. We call this sequence of n finalized states a **conclusion proof**, which we will write $\boldsymbol{\sigma}^*$.

8.2 The Consensus Game

The Consensus Game is an important ForceMove application, which we will use heavily in the rest of the paper due to its special properties regarding outcome finalizability. In this section we introduce transition rules and explore these properties.

Like all ForceMove applications, the transition rules for the Consensus Game are specified by a game library, L_C, which defines the transition function, t_{L_C}, in terms of the turn number, i, the balances, β and the game data δ. $\delta = (j, x)$, where j is the *consensus counter* and x is the *proposed balances*.

$$\begin{aligned} t_{L_C}(i, \beta, (j, x), \beta', (j', x')) \Leftrightarrow & [(j = n - 1 \wedge j' = 0 \wedge \beta' = x = x') \vee \\ & (j < n - 1 \wedge j' = j + 1, \beta = \beta', x = x') \vee \\ & (j' = 0, \beta = \beta')] \end{aligned}$$

For a given β, the consensus counter will increase from $0 \ldots (n-1)$ as the participants sign off on the new balances. Once all participants have signed, consensus has been reached and the channel's balances are updated.

8.3 Proofs of Correctness

- what do we need to prove? - we gave the algorithms in terms of universally finalizable states. - we now need to find the intermediate states to move betweenthem

- if we two universally finalizable states, σ, σ' - in a consensus game channel with n participants - that are value preserving for all participants - then it is possible to find a set of n state updates that preserve value for all participants and which move from σ and σ'

$$\begin{aligned}
\sigma^{i}(\beta,(0,\beta)) &\approx [\beta]_P \\
\sigma^{i+1}(\beta,(0,\beta')) &\approx \{\beta'\}_{p_0}[\beta]_{p_1,\ldots,p_{n-1}} \\
\sigma^{i+2}(\beta,(1,\beta')) &\approx \{\beta'\}_{p_0,p_1}[\beta]_{p_2,\ldots,p_{n-1}} \\
&\vdots \\
\sigma^{i+n-1}(\beta,(n-1,\beta')) &\approx [\beta',\beta]_{p_{n-1}} \\
\sigma^{i+n}(\beta',(0,\beta')) &\approx [\beta']_P
\end{aligned}$$

8.4 Virtual Channels on Turbo

8.5 Payouts to Non-Participants

8.6 Possible Extensions

8.7 On-chain Operations Code