Nitro Protocol

Tom Close

February 3, 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, allows for off-chain payments on the Bitcoin network. The payments make use of hashed timelocked contracts (HTLCs), which can be thought of as payments that are conditional on the revelation of a hash pre-image 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 Our State Channel Model

- introduce our model for networks of channels - both turbo and nitro work on this

- build up the foundations

- basic balances / toy system / informal description - proving things

3.1 A System of Balances

Start with a simple system of balances and show how to build it into a system of state channels

- balances + addresses - [diagram: adjudicator, including circle/square diagram]

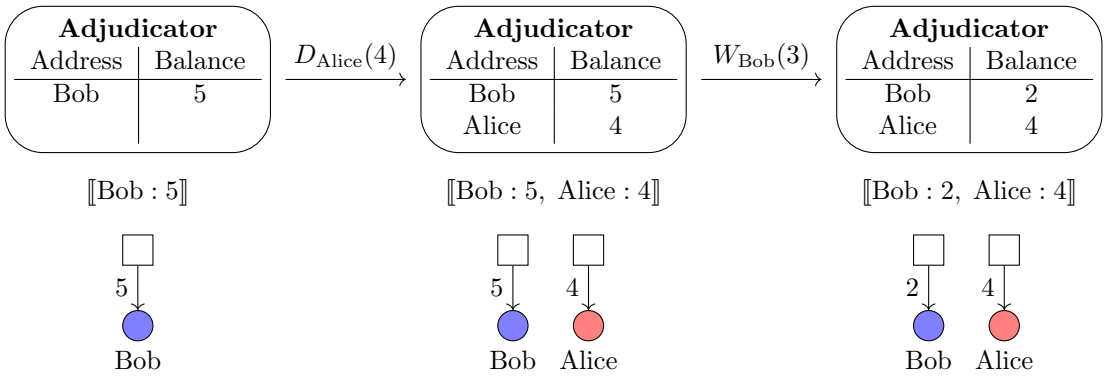


Figure 1: Cool, huh?

- coins - deposit + withdraw

- so far this is pointless

3.2 Channel Operation

- state channel participants and addresses - operation + finalization

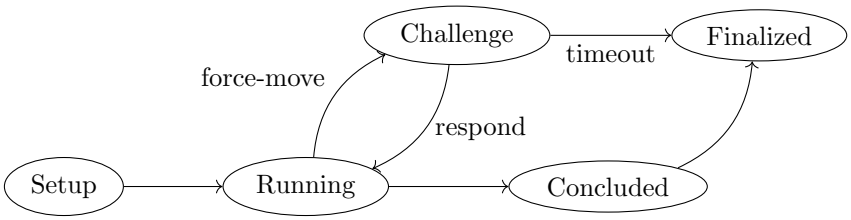


Figure 2: Channel Operation

3.3 Redistribution

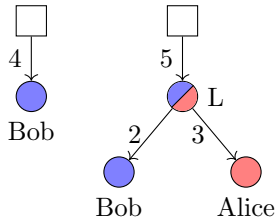
-

- outcomes (in the abstract) - outcome is what gets put on chain - outcomes used for redistribution

Adjudicator

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

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



Adjudicator

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

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

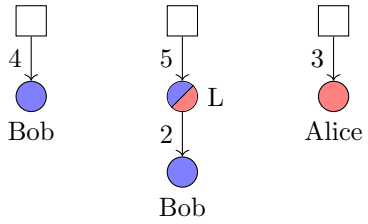


Figure 3: Cool, huh?

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 - what can I definitely finalize? - what can I definitely redistribute?

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

- definition: system state - updates with a statement. Statements are atomic. We don't allow cross-channel updates.

- definition: value - definition: a channel being funded - definition: offloaded

- paper will be concerned with funding constructions - when evaluating a construction : - that a construction funds a channel - that we can build the construction in a safe manner with single statement updates - building is important. update channels independently

- this guarantees that if any participant stops cooperating at any point, the other participants don't lose out - present protocols. If any participant stops at any point, need to know that all other participants can recover their funds - by "in a safe manner" we mean to find a sequence of states where the value is fixed

- safety = constant value at each step

- definition: safe transition

- present a construction - show that it funds a given channel - show that there's a safe path to it from a known safe starting point

- how do we do this? it's hard to write these states down. - it's also hard to talk about unbeatable strategies - three key parts: finalizable outcomes, the consensus game, redistribution results 1. write a construction in terms of finalizable outcomes of consensus game channels 2. apply results about redistribution in turbo / nitro to argue that the construction funds a channel 3. show how you can construct a safe path using the consensus game

In fact we will only do 1.

4.1 Finalizable outcomes

- the rules of strategies

- how to calculate value - this paper is about constructions that fund channels

4.2 Operating vs. Funding State Channels

At the heart of our approach lies the decision to make the funding of a state channel independent from its operation.

We view a state channel as a device for allowing a fixed set of participants to determine how a set of shared assets should be apportioned between them. We refer to the set of instructions that determine how the funds should be apportioned as the **outcome** the state channel. We define the **operation** of a state channel to be the process by which the channel reaches an outcome. The **funding** of a state channel is the method of ensuring that the assets exist and that they will ultimately be distributed according to the channel's outcome.

The operation specification also defines the rules surrounding on-chain challenges and the various ways to respond to them. The ForceMove protocol is an example of a protocol that specifies the operation of a state channel. In this paper, we will not specify the operation of state channels, but will use ForceMove as an example where required.

Specifying how state channels are funded involves specifying how funds are held in escrow on the chain, how they can be deposited and how they can be claimed according to the outcome of a state channel. As we will see in this paper, it can also involve specifying the rules for how the outcomes of multiple channels interact, which enables channels to be funded and defunded without on-chain operations. In the ForceMove paper, a *SimpleAdjudicator* was specified, which only allows for direct channels between two participants.

Decoupling the funding of a channel from its operation has several advantages. Provided that the channel is funded, its operation is completely independent from the other channels in the network. The source of funding for a channel can even change from off-chain to on-chain, all without interrupting the operation of that channel. Overall, by forcing channels to operate independently and to only be coupled through funding relationships, it becomes far easier to reason about the behaviour of any applications running within a state channel.

4.3 Addresses and Coins

A state channel is a protocol followed by a set of **participants**, each defined by a unique cryptographic address. The private keys corresponding to these addresses are used to sign updates to the channel. We assume that the signature scheme is unforgeable, so that only the owner of the address has the capability to sign states as that participant.

Each channel has a **channel address** which 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. We assume that the hashing algorithm is cryptographically secure, so that it impossible for two different sets of participants to create a channel with the same address. We also assume that the signature scheme and hashing algorithm together make it impossible to create a channel address that is the same as a participant address: we call this the 'no collision' assumption. In practice, we accept that these statements will not be absolute but instead will hold with high probability.

A state channel ultimately determines the quantity of a given asset that each participant should receive. The format that the asset quantity takes is an important consideration for a state channel. Blockchains typically have a max integer size, M, meaning that a state channel on a single asset has asset quantites in $\mathbb{Z}_M$, so that quantities above M overflow. Similarly the quantities for a state channel on two assets takes values in $\mathbb{Z}_M \times \mathbb{Z}_M$. There are many other possibilities here, including having state channels governing an arbitrary set of assets. In this paper, we will simplify the explanation by only considering state channels on a single asset, taking quantity values in $\mathbb{Z}^+$, thereby explicitly ignoring integer overflow issues. We will refer to this asset as 'coins'.

4.4 Depositing, Holding and Withdrawing

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 smart contract, which we will refer to as the **adjudicator**. In practice, the adjudicator functionality could be split across multiple smart contracts.

We say that χ **holds** x, in the case where there is a quantity of x coins locked on-chain against χ's address. We write this statement $[\![\chi : x]\!]$, where the double brackets $[\![\ ]\!]$ indicate that the statement refers to state on the chain. Note that the only information stored on-chain is the channel address and the quantity of the asset held for it; all other information resides in the off-chain states held by the participants and is only visible on-chain in the case of a dispute.

The **deposit** operation, $D_\chi(x)$, is an on-chain operation used to assign x coins to channel χ. There are no restrictions on who can deposit coins into a channel, but the transaction must always include a transfer of x coins into the adjudicator.

$$D_\chi(x)[\![\chi : y]\!] = [\![\chi : (x + y)]\!] \tag{1}$$

In order to distribute the coins it holds, a state channel, χ, must have one or more mechanisms for finalizing its outcome, Ω, on-chain. This finalization must be done in a way that ensures that at most one outcome can be finalized for each channel. In ForceMove, outcomes are finalized either via an unanswered challenge or by the presentation of a *conclusion proof* - a special set of states signed by participants indicating that the channel has concluded, thus allowing them to avoid the challenge timeout when withdrawing their funds. We write $[\![\chi \mapsto \Omega]\!]$ to represent the situation where the outcome Ω for channel χ has been finalized on-chain.

Once an outcome is finalized on-chain, it can be used to transfer coins between addresses, through the application of one or more on-chain operations. For example:

$$T_{\chi,A}(x)[\![\chi : (a + b) \mapsto \Omega]\!] = [\![\chi : a \mapsto \Omega', A : b]\!] \tag{2}$$

We will explain this example in further detail in the sections on Turbo and Nitro. In equation (2), A could be either a channel address or a participant address.

The **withdrawal** operation can be used to withdraw coins held at address A by any party with the knowledge of the corresponding private key. Note that the signature requirement coupled with the no-collision assumption means it is not possible to withdraw from a channel address. If $x \leq x'$ then

$$W_A(x)[\![A : x']\!] = [\![A : (x' - x)]\!] \tag{3}$$

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 [1].

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

In practice, the adjudicator api could allow multiple operations to be executed in a single blockchain transaction. For example, in the ForceMove SimpleAdjudicator funds are withdrawn in a way such that the intermediate state, where funds are held against a participant address, is never persisted on-chain.

4.5 The Value of a State

The utility of a state channel comes from the ability to transfer the value between participants without the need for an on-chain operation. In order to reason about state channels, we therefore need to understand how this transfer of value works.

The word 'state' is very overloaded in the world of state channels. In what follows we will need to distinguish between the state of an individual channel and the entire state of all channels plus the adjudicator. We will call the latter the **network state** and denote it with the symbol Σ.

We define the **value**, $\nu_A(\Sigma)$, of a network state Σ for participant A, to be the largest x such that A has an unbeatable strategy to extract [2] x coins from the adjudicator. The unbeatable strategy can involve signing (or refusing to sign) states off-chain, as well as applying one or more on-chain operations. The strategy might have to adapt based on the actions of other players but regardless of the actions they take, it should still be possible for A to extract x coins.

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.
3. **Transactions are free**: we ignore the cost of gas fees when calculating the value of a state.

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.

Throughout this paper we will present sequences of states that interpolate between a start state and a target state, whilst preserving value for all participants.

[2] By *extract* we mean to withdraw x more coins than were deposited in the execution of the strategy.

We will make the argument that, as the value is constant[3], participants will be willing to transition between these states. We call these transitions **safe transitions**.

If we were considering transactions fees here, we would need to argue this principle more carefully: with transaction fees, some of the transitions we see as value preserving here would actually involve moving to a state of slightly lower value. We assume that the utility gained in being able to open and close channels off-chain overcomes this issue in practice. In general, modelling the effect of gas fees on state channel networks is an interesting and important area of research but falls outside the scope of this paper.

4.6 Finalizable and Enabled Outcomes

In this paper, we are not generally concerned with the operation of state channels. However, in reasoning about channels that fund other channels, we do need to be able to talk about the states of these channels. It turns out that it is enough to characterise states in terms of the outcomes they allow the participants to register, which allows us to remain agnostic to the precise operation of the channels. In this section we develop the tools for characterising states in this way.

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.

$$[\chi \mapsto \Omega]_A \xrightarrow{\text{A's unbeatable strategy}} [\![\chi \mapsto \Omega]\!] \tag{4}$$

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

1. Participant p is the unique participant with one or more finalizable outcome(s), $\Omega_1, \ldots, \Omega_m$. We write this $[\chi \mapsto \Omega_1, \ldots, \Omega_m]_p$.
2. There are at least two participants, $P = \{p_1, \ldots, p_m\}$, who share the same finalizable outcome, Ω. We write this $[\chi \mapsto \Omega]_{p_1,\ldots,p_m}$.
3. 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

[3]The one exception is any transition involving a deposit, where we assume that a participant depositing x coins into the network, will proceed if the value of the resulting state for them increases by x.

of the protocols we present make use of the final situation, where no participant has a finalizable outcome, and we believe this state should generally be avoided.

In the special case where the outcome of a channel is finalizable by all its participants, we say that the outcome is **universally finalizable**. This happens at two points of every ForceMove channel's 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 to prevent from being finalized. We write the set of enabled outcomes for p as $[\chi \mapsto \Omega_1 \dots \Omega_m]_p$.

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.

4.7 The 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 give a complete definition of the consensus game in the appendix.

The rules of the consensus game ensure that it reaches a universally finalizable outcome at least once every n turns, for a channel with n participants. At a very high level, the consensus game provides a mechanism for moving from one universally finalizable outcome, Ω_1, to another, Ω_2. In order for this to happen, one participant proposes the new outcome, Ω_2, and then every other

[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.

participant must sign off on it. If any participant disagrees, they can cancel the transition.

The consensus game has some desirable properties in terms of enabling value preserving transitions between network states, which are useful when proving the correctness of state channel network constructions. Because of this, consensus game channels will feature heavily in Turbo and Nitro protocols.

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:

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

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:

$$[\![L : 10]\!], [L \mapsto (A : 2, B : 3, \chi : 5)]_{A,B} \tag{6}$$

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:

$$T_{L,\chi}(5)[\![L : 10 \mapsto (A : 2, B : 3, \chi : 5)]\!] = [\![L : 10 \mapsto (A : 2, B : 3), \chi : 5]\!] \tag{7}$$

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:

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

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

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

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

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

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 $x = a + b + c$:

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

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

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

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

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

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 $x = a + b + c$:

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

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.

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

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

$$D_L(a')[\![L : x]\!] = [\![L : x + a']\!] \tag{16}$$

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

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

5.1.4 Partial checkout 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:

$$[\![L : x]\!],\ [L \mapsto (A : a + a', B : b, \chi : c)]_{A,B} \tag{18}$$

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.

$$[L' \mapsto (A : a, B : b, \chi : c)]_{A,B} \tag{19}$$

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

$$[L \mapsto (L' : a + b + c, A : a')]_{A,B} \tag{20}$$

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:

$$[\![L : x \mapsto (L' : a + b + c, A : a')]\!] \tag{21}$$

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

$$T_{L,A}(a')[\![L : x \mapsto (L : a + b + c, A : a')]\!] = \\ [\![L : x - a' \mapsto (L' : a + b + c),\ A : a]\!] \tag{22}$$

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

$$T_{L,L'}(a + b + c)[\![L : x \mapsto (L : a + b + c),\ A : a']\!] = \\ [\![L' : a + b + c,\ A : a']\!] \tag{23}$$

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.

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.

$$C_{G,B}(5)[\![G : 5 \mapsto (L|B), L \mapsto (A : 5, B : 5)]\!] = \\ [\![G \mapsto (L|B), L \mapsto (A : 5), B : 5]\!] \quad (24)$$

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.

$$[\![L : x, L' : x]\!],\ [L \mapsto (A : a, I : b)]_{A,I},\ [L' \mapsto (B : b, I : a)]_{B,I} \quad (25)$$

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 4.

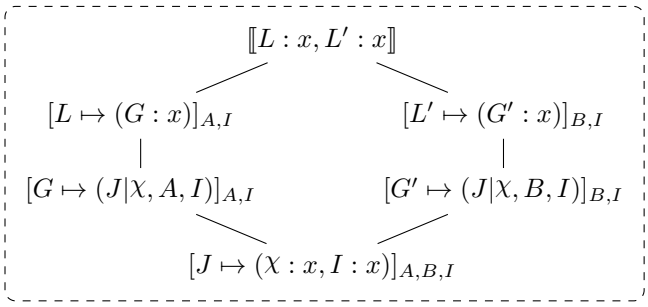


Figure 4: Virtual channel construction

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.

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:

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

   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:

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

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

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

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:

$$[\![L : x \mapsto (G : x),\ L' : x \mapsto (G' : x), G \mapsto (J|\chi, A, I), \\ G' \mapsto (J|\chi, B, I),\ J \mapsto (\chi : x, I : x)]\!] \quad (29)$$

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

$$[\![G : x \mapsto (J|\chi, A, I),\ G' : x \mapsto (J|\chi, B, I),\ J \mapsto (\chi : x, I : x)]\!] \quad (30)$$

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

$$[\![G \mapsto (J|\chi, A, I),\ G' : x \mapsto (J|\chi, B, I),\ J \mapsto (I : x),\ \chi : x]\!] \quad (31)$$

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

$$[\![G \mapsto (J|\chi, A, I),\ G' \mapsto (J|\chi, B, I),\ \chi : x,\ I : x]\!] \quad (32)$$

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 (25):

$$[\![L : x, L' : x]\!] \quad (33)$$
$$[L \mapsto (A : a, I : b)]_{A,I} \quad (34)$$
$$[L' \mapsto (B : b, I : a)]_{B,I} \quad (35)$$

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

$$[\chi \mapsto (A : a, B : b)]_{A,B} \tag{36}$$

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

$$[J \mapsto (A : a, B : b, I : x)]_{A,B,I} \tag{37}$$
$$[G \mapsto (J|\chi, A, I)]_{A,I} \tag{38}$$
$$[G' \mapsto (J|\chi, B, I)]_{B,I} \tag{39}$$

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

$$[L \mapsto (G : x)]_{A,I} \tag{40}$$
$$[L' \mapsto (G' : x)]_{B,I} \tag{41}$$

5. Switch J over to fund χ:

$$[J \mapsto (\chi : x, I : x)]_{A,B,I} \tag{42}$$

We give a visual representation of this procedure in figure 5.

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:

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

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:

$$[J \mapsto (A : a', B : b', I : x)]_{A,B,I} \tag{44}$$

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

$$[L \mapsto (A : a', I : b')]_{A,I} \tag{45}$$
$$[L' \mapsto (B : b', I : a')]_{B,I} \tag{46}$$

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

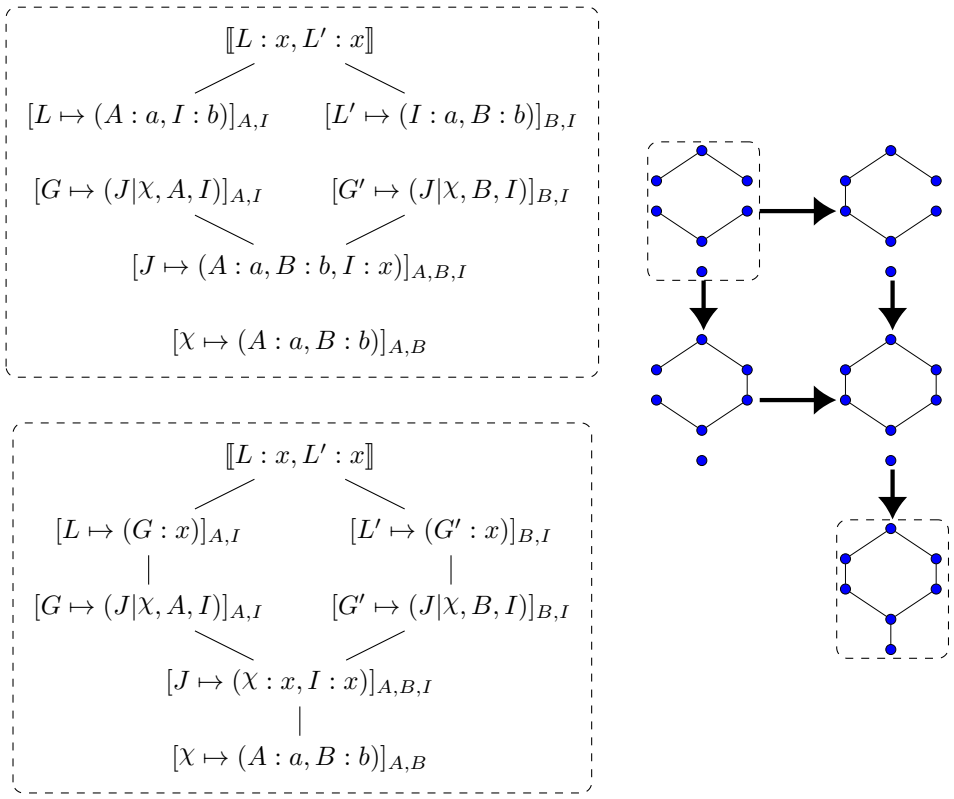


Figure 5: Opening a virtual channel

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 [].

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

`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 particpant.
`r`	`bytes32`		
`s`	`bytes32`		

Table 1: ForceMove state format

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, $t_L(i, \beta, \delta, \beta', \delta') \rightarrow \{T, F\}$. 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