# Nitro Protocol

Tom Close

January 25, 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 be fundone 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.

'to fund' or 'to be funded by'

## 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.

- state channels allow state updates 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.

complete this thought

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

Ledger channels allow multiple channels between a given set of participants to be supported by a single on-chain deposit. Importantly, these channels still update independently to one another, even though they are funded by the same deposit.

As an example of what ledger channels enable, suppose Alice and Bob want to play a game of chess and that they already have an existing ledger channel, $L$, open between them. The winner of the game of chess should receive 2 coins from the loser and $L$ currently holds 5 of Alice's coins and 5 of Bob's.

| Ledger Channel [v1] | |
|---|---|
| Alice | 5 |
| Bob | 5 |

Alice and Bob proceed by preparing a new channel for the chess game with the appropriate starting state. Ordinarily, the chess game would now require an on-chain transaction. Instead, Alice and Bob then update the ledger channel to allocate funds to these games.

| Ledger Channel [v2] | |
|---|---|
| Alice | 3 |
| Bob | 3 |
| Chess | 4 |

| Chess [v1] | |
|---|---|
| Alice | 2 |
| Bob | 2 |

Once the funds are allocated, they are free to play the game of chess. Updates to the chess channel are independent from updates to the ledger channel $L$ and to any other sub-channels that are potentially funded by it. Alice wins the chess game, so the final state in the chess channel allocates all the funds to her.

| Ledger Channel [v2] | |
|---|---|
| Alice | 3 |
| Bob | 3 |
| Chess | 4 |

| Chess [FINAL] | |
|---|---|
| Alice | 4 |
| Bob | 0 |

To close the chess channel off-chain, Alice and Bob update the state of the ledger channel to absorb the outcome of the game.

| Ledger Channel [v3] | |
|---|---|
| Alice | 7 |
| Bob | 3 |

Virtual channels go one step further than ledger channels, allowing a state channel to be opened between two participants who do not share an on-chain deposit, but instead have a counterparty in common, with whom they both have a channel open with.

As an example, suppose that Alice wants to open an $\{A : 5, B : 5\}$ chess game with Bob, but they do not share an existing ledger channel. Alice does, however, have an $\{A : 5, C : 5\}$ ledger channel open with Ingrid and Bob and Ingrid have a $\{B : 5, C : 5\}$ ledger channel open too.

there are two problems here: 1. the outcome notation here doesn't match the notation used later, and 2. the notation for an outcome isn't ever defined

By using a virtual channel, Alice and Bob may open their chess game, using their existing ledger channels with Ingrid, without the need for any on-chain transactions. Suppose that Alice wins all of the funds in the channel, so that the final outcome is $\{A : 10, B : 0\}$. Alice and Bob can also close their chess game off-chain, rebalancing through Ingrid, so that the final state of the ledger channels is $\{A : 10, C : 0\}$ and $\{B : 0, C : 10\}$.

In this interaction, Ingrid contributed 10 of her coins at the beginning, which had to remain locked while Alice and Bob were playing. At the end of the interaction, Ingrid still had 10 coins but they were distributed differently between the her ledger channels. By locking up her coins for a period of time, she enabled Alice and Bob to play chess entirely off-chain, including the setup and conclusion phases. Although all this occured off-chain, Alice and Bob received security assurances similar to if the entire interaction had been on-chain.

One good way of summarising the differences between these three different approaches is to look at the number of on-chain deposits required in each of them:

- **Direct channels**: one state deposit per channel.
- **Ledger channels**: one state deposit per group of interacting participants, regardless of how many channels they participate in together.
- **Virtual channels**: one state deposit per user, regardless on how many channels they participate in.

Virtual channels hold the biggest potential in terms of scaling, requiring only one on-chain deposit per participant.

# 2 Existing work

- lightning network - raiden - in production, payments only

| Operation | Funding |
|---|---|
| States | Deposits |
| Transitions | Withdrawals |
| Challenges | |
| Responses | |

- celer network proposes using

- counterfactual using counterfactual instantiation - high level overview about how to implement meta channels details are not presented in the paper

- perun virtual channels

### 2.1 Our contribution

- virtual and ledger channels - channels are unrestricted - no time limits or special update rules due to being in a virtual channel

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

## 3 State Channel Background

In this section, we introduce the state channel concepts necessary to understand the rest of the paper.

### 3.1 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.

Specifying the operation of a state channel involves specifying the format of the states and the update rules that define the allowed transitions between these states. 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, all funding is performed by the *SimpleAdjudicator*, which only allows for direct channels between participants.

Perhaps it's better to say that in the ForceMove paper, a SimpleAdjudicator is specified, which only allows a single direct channel between two participants. (My point is that the ForceMove paper is about the specification of the operation of a state channel, and not really the funding of it.)

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.

## 3.2 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'.

"it possible for any participant to force a distinct channel address, by using a unique nonce"

### 3.3 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 $\chi$ **holds** $x$, in the case where there is a quantity of $x$ coins locked on-chain against $\chi$'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 $\chi$. 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, $\chi$, must have one or more mechanisms for registering its outcome, $\Omega$, on-chain. This registration must be done in a way that ensures that at most one outcome can be registered for each channel. In ForceMove, outcomes are registered 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 $\Omega$ for channel $\chi$ has been registered on-chain.

It seems like 'finalized' and 'registered' mean the same thing. If so, this should be made clear – perhaps you should just used 'finalized'

Once an outcome is registered 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.

check that this is explained

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 only possible to withdraw from a participant address. If $x \leq x'$ then

... if you know the private key?

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

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].

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.

### 3.4 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 $\Sigma$.

We define the **value**, $\nu_A(\Sigma)$, of a network state $\Sigma$ 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.

Isn't this assumption totally false in practice? Do you mean 'There is an $\epsilon > 0$ such that for any $t$, it is possible ...'

The first assumption sidesteps issues of censorship, chain congestion and timing considerations around the creation of blocks. In practice, this assumption should

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

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

hold if $\epsilon$ 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. 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.

### 3.5 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, $\Omega$, is **finalizable** for participant $A$, if $A$ has an unbeatable strategy for registering this outcome in the adjudicator. We use the notation $[\chi \mapsto \Omega]_A$, to represent a state of a channel, $\chi$, where the outcome, $\Omega$, 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 $\chi$ 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, $\Omega$. We write this $[\chi \mapsto \Omega]_{p_1,\ldots,p_m}$.

[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 $x$.

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 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 exists.In this state, we say the channel is in the **concluded state**.

is known to each participant?

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, $\chi$, 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.

## 3.6 The Consensus Game

It's a bit weird to introduce the consensus game here, but say almost nothing about it. I think it would be hard to continue reading without specifying what properties the consensus game has that allows the imple-

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.

[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.

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, $\Omega_1$, to another, $\Omega_2$. In order for this to happen, one participant proposes the new outcome, $\Omega_2$, and then every other 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.

# 4 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 4.1.** 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 registered on-chain. As $L$ can afford 5 for $\chi$ 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.

## 4.1 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 $\chi$:

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

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 $\chi$ in the above setup. This is because either participant has the power to convert the situation above into a situation where $\chi$ 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 $\chi$ has been **offloaded**. Note that we do not need to wait for $\chi$ to complete before offloading it. The offload converts $\chi$ 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 $\chi$ 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.

In practice, with non-zero gas prices, it is ...

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

### 4.1.1 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 $\chi$ 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:

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

#### 4.1.2 Closing a sub-channel

When the interaction in a sub-channel, $\chi$, 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 $\chi$ 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 $\chi$, 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 updatethe ledger channel to include the result of channel $\chi$.

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

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

It's not really clear why a participant would be forced to update the ledger channel in this way.

#### 4.1.3 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, $\chi$. 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:

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

#### 4.1.4 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.

# 5 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 a target allocation, whose debts it will help to pay. When paying out debts, the guarantee outcome can choose to modify the payout priority order of its target allocation.

target address?

Suggested rewording in comments

We will use the notation $(L|A, B)$ for a guarantee with target $L$, which prioritizes first $A$, then $B$, then to any other addresses according to the priorities in $L$'s allocation. 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.

This is a strange definition – it seems like a guarantee must specify the priority of exactly two addresses?

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 $C_{G,A}(x)$:

- 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 $x$.

If the outcome of $L$ is not yet registered on-chain, or if $G$ cannot afford $x$ for $A$, then the operation has no effect.

check why this terminology makes sense

**Example 5.1.** In the following example, we have a guarantor channel, $G$, which holds 5 coins and guarantees $L$'s allocation, with $\chi$ as highest priority.

I think this needs some clarification, as the definition of 'can afford $x$' is almost circular. Loose suggestion in comments

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

$\chi \mapsto B$

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

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

'of an adjudicator implenenting the Nitro protocol in the appendix'

## 5.1 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 give a construction for the simplest possible virtual channel, between $A$ and $B$ through a shared intermediary, $I$.

Virtual channels between larger groups of participants are left as an exercise for the reader :)

Suppose we have a pair of ledger channels, $L$ and $L'$, with participants $\{A, I\}$ and $\{B, I\}$ respectively. Assume the following network state.

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

where $x = a + b$. The participants want to use the existing deposits and ledger channels to fund a virtual channel, $\chi$, 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 1.

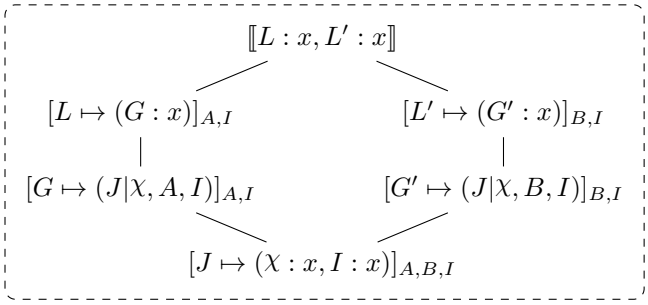


Figure 1: Virtual channel construction

We will cover the steps for safely setting up this construction in section 5.3. In the remainder of this section we will explain why this construction can be considered to fund the channel $\chi$. Similarly to the method for ledger channel construction, we will do this by demonstrating how any one of the participants can offload the channel $\chi$: converting it to an on-chain channel that holds its own funds.

In section 5.2, we will explain ...

Shouldn't you also show how the participants can jointly update

## 5.2 Offloading Virtual Channels

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

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

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

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

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

A can't finalize $L'$, only $L$ and $J$.

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

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

As $G$ has $\chi$ 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. We assume that $I$ has started by registering all their finalizable channels on-chain, followed by transferring funds from $L$ to $G$ and from $L'$ to $G'$.

This sentence seems redundant, given that the steps are laid out in detail below

1. $I$ starts by registering 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 $\chi$:

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

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

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

$$C_{G',G}(I)C_{G,\chi}(x)[\![G : x \mapsto (J|\chi, A, I), G' : x \mapsto (J|\chi, B, I), \\ J \mapsto (\chi : x, I : x)]\!] = [\![L' : x, \chi : x, I : x]\!] \quad (33)$$

This equation isn't needed, and also appears to be nonsense

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

As in Turbo, the offload is an action of last resort and virtual channels are designed to be opened and closed entirely off-chain. We now show how this can be accomplished.

This paragraph seems better suited at the end of subsection 5.1, or at the start of subsection 5.3

## 5.3 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:

It seems like this section requires a proof of safety at each step, perhaps in the index.

1. Start in the state given in equation (25):

$$[\![L : x, L' : x]\!] \tag{34}$$
$$[L \mapsto (A : a, I : b)]_{A,I} \tag{35}$$
$$[L' \mapsto (B : b, I : a)]_{B,I} \tag{36}$$

2. $A$ and $B$ bring their channel $\chi$ to the funding point:

$$[\chi \mapsto (A : a, B : b)]_{A,B} \tag{37}$$

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

$$[J \mapsto (A : a, B : b, I : x)]_{A,B,I} \tag{38}$$
$$[G \mapsto (J|\chi, A, I)]_{A,I} \tag{39}$$
$$[G' \mapsto (J|\chi, B, I)]_{B,I} \tag{40}$$

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

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

5. Switch $J$ over to fund $\chi$:

$$[J \mapsto (\chi : x, I : x)]_{A,B,I} \tag{43}$$

We give a visual representation of this procedure in figure 2.

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

1. Participants $A$ and $B$ finalize $\chi$ by signing a conclusion proof:

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

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

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

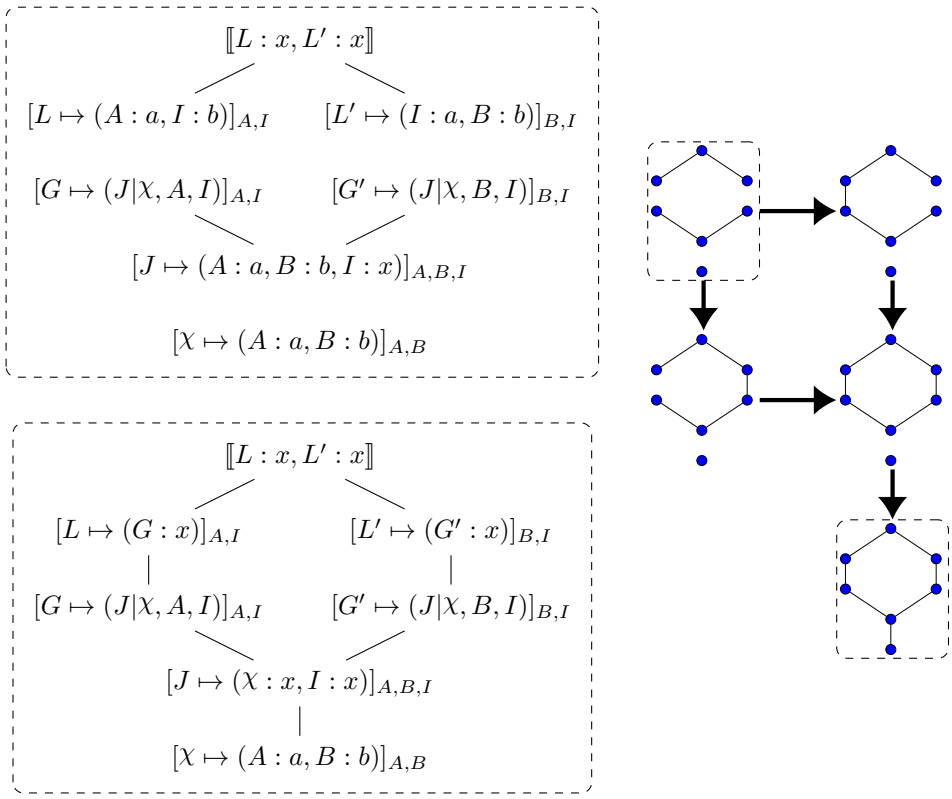


Figure 2: Opening a virtual channel

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

$$[L \mapsto (A : a', I : b')]_{A,I} \tag{46}$$
$$[L' \mapsto (B : b', I : a')]_{B,I} \tag{47}$$

4. The channels $\chi$, $J$, $G$ 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.

I feel like you should explain how to do this. It should be possible to reduce it to the partial checkout process from section 4.

# 6 Acknowledgements

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

# 7 Appendix

THIS APPENDIX IS STILL A WIP.

## 7.1 Overview of ForceMove

## 7.2 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, $\beta$ and the game data $\delta$. $\delta = (j, x)$, 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 $\beta$, 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.

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

## 7.3 Proofs of Correctness

- what do we need to prove? - we showed that

- if we two universally finalizable states, $\sigma$, $\sigma'$ - 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 $\sigma$ and $\sigma'$

## 7.4 Virtual Channels on Turbo

## 7.5 Payouts to Non-Participants

## 7.6 Possible Extensions

## 7.7 On-chain Operations Code

### 7.7.1 Overlap

```
def overlap(recipient, outcome, funding):
  for [address, allocation] in outcome:
    if funding <= 0:
      return 0;
    elif address == recipient:
      return min(allocation, funding)
    else:
      funding -= allocation

  return 0
```