Nitro Protocol

Tom Close

January 13, 2019

1 Motivation

A blockchain is a device that enables a group of adversarial parties to come to consensus over the contents of a shared ledger, without appealing to a trusted central authority. Whether using a proof-of-work or a proof-of-stake algorithm, this process has a cost both in terms of time and in terms of money. For the Ethereum blockchain, this cost results in an effective limit of around 15 transactions per second being processed on the network. When compared with the visa network, which can process on the order of 50,000 transactions per second, this limit supports the view that blockchains, in their current form, do not scale.

State channels offer a solution to the blockchain scaling problem. A state channel can be thought of as a set of updatable agreements between a fixed set of participants determining how a given set of assets should be split between them. The agreements are updated off-chain through the exchange of cryptographically signed messages between the participants according to some set of previously-agreed update rules. The assets in question are held in escrow on the blockchain, in such a way that they can only be released according to the latest agreement from the state channel. If the off-chain cooperative behaviour breaks down, for example if one party refuses to sign updates either by choice or due to unavailability, any of the parties can reclaim their share of the assets by presenting the latest agreement to the chain. An exit game is required to give other parties the opportunity to present a later state, but all parties have the guarantee that they can reclaim their share of the funds within some finite time.

In addition to the increased throughput, state channels bring instant finality to transactions: the moment a channel update is received the participant knows that the assets transferred are now assigned to them. They cannot access the assets immediately but they have the power to prevent any other party claiming those assets in the future. The only requirement is that participants need to be live enough to engage in the exit game if an opponent attempts to exit an earlier state. In practice, the requirement here is to check the chain periodically

- at time intervals that are less than, but on the same order of magnitude as, the challenge duration in the exit game.

As the channel updates are created and exchanged off-chain, the channel-update throughput is limited only by the speed of constructing, signing and broadcasting the update. Given that channel updates only needed to be communicated between participants and the system is therefore highly parallelizable, it is difficult to put a bound on the total transaction throughput of a system of state channels. In practice the system is only limited by the on-chain operations required to move assets when opening and closing channels.

In this paper, we allow channels to be opened and closed off-chain. We enable the construction of efficient state channel networks.[TODO]

2 Existing work

Lightning and raiden - payment channel networks - directional

Celer - directional but with multiple paths - general conditions

Perun - virtual payment channels (different from HTLCs) - but using a validity time - and then state channel networks - constrained [Connext - have removed the time limit with a trade-off of trusting the hub - TODO: check this]

Counterfactual - thinks about counterfactual instantiation - also app instance on top, which we collaborated on - possible to do meta-channels, but the details are not given in the paper and are still being finalized

ForceMove

Our contribution: - like perun without the time limit Unconstrained subchannels

3 State Channel Background

3.1 Funding vs Operation

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 split between them. We refer to the set of instructions that determine how the funds should be split 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 outcome is honoured on-chain.

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. Specifying the operation also involves defining the rules surrounding on-chain challenges and the 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, instead leaning on ForceMove 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 was done by the *SimpleAdjudicator*, which only allowed for direct channels between participants.

Decoupling the funding of the channel from its operation has several advantages. Provided that a channel is funded, its operation is completely independent from the other channels in the system. 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 only be coupled through funding relationships, it becomes a lot easier to reason about the behaviour of any applications running within a state channel.

3.2 Participants, Addresses and Assets

A ForceMove state channel belongs to 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. 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 the max 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 on 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'.

3.3 Depositing, Holding and Withdrawing Coins

In order to have value, a state channel system 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 and deposits 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 $[\![\ ]\!]$ indicates that the statement refers to state on the chain. Note that in ForceMove, 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 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 - only that the transaction must include a transfer of x coins into the adjudicator.

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

In order to distribute the coins it holds, a state channel, χ, must have one or more mechanisms for registering its **outcome**, Ω_χ, 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 skip the challenge timeout. We write $[\![\chi \mapsto \Omega]\!]$ to represent the situation where the outcome Ω for channel χ has been registered on-chain.

- holding nothing

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:

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

The specification of the precise format for the outcome, Ω_χ, and the operations, O, will be given in the sections on Turbo and Nitro. In the equation above, 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 only possible to withdraw from a participant address. If $x \leq x'$ then

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

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 operations O and W do not need to be separate blockchain transactions. For example, in the ForceMove SimpleAdjudicator, funds are withdrawn directly using α_χ and Ω_χ; the intermediate state, α_A, where funds are held against a participant address never exists 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 **system state** and usually denote it with the symbol Σ.

We define the **value**, $\nu_A(\Sigma)$, of a system state Σ for participant A, to be the largest x such that A has an *unbeatable strategy* to extract x coins from the adjudicator, where by extract we mean to withdraw x coins more than any we needed to deposit to execute the strategy. 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.

In order to make the concept of an unbeatable strategy more tangible, 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 $t + \epsilon$. This assumption sidesteps issues of censorship, chain congestion and timing considerations around the creation of blocks.

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

In practice, this assumption should be reasonable, provided that ϵ is sufficiently large.

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. This assumption means that strategies that rely on executing a given transaction on-chain before someone else executes a different one are not allowed.

3. **Transactions are free**: we ignore the cost of gas fees when calculating the value of a state. Modelling the effect of gas fees on state channel networks is an interesting and important area of research but falls beyond the scope of this paper.

Although the main utility of state channels comes from transitions where the value changes, in this paper we will be looking primarily at state transitions where the value of the system remains the same. We will use the fact that certain transitions preserve value for all participants, to allow us to prove the correctness of operations involving the construction of state channel networks.

In particular, we will make use of the **principle of value equivalence**: if two system states hold the same value for all participants, then we assume that all participants will be willing to transition between them. The one exception is any transition involving a deposit, where we assume that a participant depositing x coins into the system, will proceed if the value of the resulting state to them increases by x.

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 issues in practice.

When analysing Turbo and Nitro, we will often need to calculate the value of a given system state for a participant. When doing this, it will be useful to break the strategies down into two stages: the strategy for registering one or more outcomes on-chain, followed by the strategy for performing operations on those outcomes on-chain. We will focus on the former stage in the next section; the latter stage will be dealt with separately in the sections for Turbo and Nitro.

3.5 Finalizable and Enabled Outcomes

In the previous section, we defined the value of a system in terms of an unbeatable strategy for withdrawing a certain total. In this section, we will focus on one part of this strategy: the ability to register a given outcome in the adjudicator.

We say an outcome, Ω, 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, χ, where the outcome, Ω, is finalizable by A.

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

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

1. No participants apart from p have a finalizable outcome. Participant p has one or more finalizable outcome(s), $\Omega_\chi^{(1)}, \ldots, \Omega_\chi^{(m)}$. We write this $[\Omega_\chi^{(1)} \ldots \Omega_\chi^{(m)}]_p$.
2. There are at least two participants, $P = \{p_1, \ldots, p_m\}$, who share the same finalizable outcome, Ω_χ. We write this $[\Omega_\chi]_{p_1,\ldots,p_m}$.
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. We will see that all these possibilities naturally occur in sensible state channel protocols except the last case, which usually means something has gone wrong.

Example 3.1. One example of when a finalizable outcome occurs is for the next mover in a ForceMove channel. In ForceMove, participants take turns to sign states, so that in a channel with n participants, p_i can only sign state σ_m if $i = m\%n$. If p_i has just received σ_{m-1}, with *current outcome*[2] Ω_χ, then Ω_χ is a finalizable outcome for p_i. There are a couple of strategies here: one is to refuse to sign σ_m, trigger a force-move on σ_{m-1} and wait for the timeout. Another is to transition to transition to the conclude state, to allow a conclusion proof to be created for ω, so that the participants can avoid waiting for the timeout [3].

Example 3.2. A ForceMove channel with two different conclusion proofs are held by two different participants is an example of a situation where there are no finalizable outcomes for any participant. Each conclusion proof can be immediately finalized on-chain, with no opportunity for challenge. Therefore, the first conclusion proof to be presented to the adjudicator wins. Due to our decision to allow front-running, there is no strategy that can ensure that one conclusion proof will always hit the adjudicator first. As discussed above, this is not a desireable situation. This is the reason that participants should never sign two conclusion proofs with different outcomes.

[2]The current outcome is stored as part of the state in ForceMove.

[3]The fact that the first strategy exists is why ForceMove allows a transition to conclude from any state, to allow participants to attempt the second strategy and save everyone some time

If a participant has no finalizable outcomes, their analysis of the system 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 $[\Omega_\chi^{(1)} \dots \Omega_\chi^{(m)}]_{(p)}$.

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, due to the property that any participant can force an outcome within a finite time, that if a participant has only enabled a single outcome, that outcome must be finalizable for them.

Example 3.3. All ForceMove channels start with a setup phase of $2n$ states, where n is the number of participants. During this phase, the framework transition rules prevent the current outcome, Ω_χ, from changing. At the midpoint, when state σ_n has just been broadcast, the outcome Ω_χ is finalizable for every participant, i.e. we have $[\Omega_\chi]_{p_1,\dots,p_n}$. The outcome Ω_χ is finalizable because, due to the special transition rules, it is the only outcome that each participant has enabled by the states they have signed so far. As we will see later, this property of a ForceMove channel is important when opening and closing channels.

We will finish this section on finalizable outcomes by talking about **universal finalizability**. A state channel with participants $P = \{p_1, \dots, p_n\}$ is said to be in a universally finalizable state if there is an outcome, Ω_χ, that is finalizable for every participant, i.e. $[\Omega_\chi]_{p_1,\dots,p_n}$. We will sometimes write this state using the shortened notation $[\Omega_\chi]_P$.

We have already come across two important examples of times when a ForceMove state channel gets into a universally finalizable state:

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**.

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].

[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.

3.6 Value Preserving Consensus Transitions

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.

At a very high level, the consensus game provides a mechanism for moving from one universally finalizable outcome, $\Omega_\chi^{(1)}$, to another, $\Omega_\chi^{(2)}$. In order for this to happen, one participant proposes the new outcome, $\Omega_\chi^{(2)}$, and then every other participant must sign off on it. If any participant disagrees, they can cancel the transition. This process has the important property that throughout its operation no participant enables any outcome other than $\Omega_\chi^{(1)}$ and $\Omega_\chi^{(2)}$. We call this a **consensus transition** between the two outcomes.

In the presentation of Turbo and Nitro, we will often use the fact that, if you use the consensus game transition to move between two system states with the same value to a participant, then all the intermediate states also have the same value to that participant. This is the key observation that allows us to prove that we can open and close channels off-chain.

Value Preserving Consensus Transitions (VPCTs): If I have two system states, Σ and Σ', which only differ in the state of a consensus game chanel, χ, with participants P and universally enforceable outcomes $[\Omega_\chi]_P \in \Sigma$ and $[\Omega'_\chi]_P \in \Sigma'$, and for some participant $p \in P$, $\nu_p(\Sigma) = \nu_p(\Sigma') = x$, then applying a consensus game transition gives a sequence of system states $\Sigma = \Sigma_0, \ldots, \Sigma_m = \Sigma'$ where $\nu_p(\Sigma_i) = x$ for all Σ_i.

Proof (sketch): As $\nu_p(\Sigma) = \nu_p(\Sigma) = x$ and Σ and Σ' only differ in χ, we know that if p can register either outcome Ω_χ or Ω'_χ on-chain, then they can obtain value x. The consensus transition allows the participants to transition from outcome Ω_χ to Ω'_χ without any participant enabling any other outcome. Therefore at all points, p is capable of registering one of those outcomes on-chain (though in general they cannot choose which). Therefore, at all points, the value of the system to p is x.

4 Turbo

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 the total, v_i, represents that amount of coins due to the 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 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 $T_{A,B}(x)$: (a) reduces the funds held by A by x, (b) increases the funds held by B by x, and (c) 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 4 for χ the following transfer operation is successful:

$$T_{L,\chi}(4)[\![L{:}10 \mapsto (A:3, B:2, \chi:5)]\!] = [\![L{:}6 \mapsto (A:3, B:2, \chi:1), \chi{:}4]\!]$$

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

4.1 Ledger Channels

A **ledger** channel is a channel who 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 on-chain to participants A and B and channel χ:

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

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 above set of states into a situation where χ is funded on-chain:

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

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.

In Turbo, all ledger channels are regular ForceMove channels running the Consensus Application.

4.1.1 Opening a sub-channel

The value of a ledger channel comes 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:
   - $[\![L{:}x]\!],\ [L \mapsto (A : a, B : b)]_{A,B}$
2. A and B create their sub-channel χ and progress it to the funding point. We assume that $a' \leq a$ and $b' \leq b$:
   - Create channel χ: $[\chi \mapsto (A : a', B : b')]_{A,B}$
3. Update the ledger channel to fund the sub-channel:
   - Update L: $[L \mapsto (A : a - a', B : b - b', \chi : a' + b')]_{A,B}$

4.1.2 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:
   - $[\![L{:}x]\!],\ [L \mapsto (A : a, B : b, \chi : c)]_{A,B}$, where $x = a + b + c$.
2. The next step is for A and B to concluded channel χ, leaving the channel in the concluded state.
   - Update channel χ: $[\chi \mapsto (A : a', B : b')]_{A,B}$, where $a' + b' = c$.
3. The participants then update the ledger channel to include the result of channel χ.
   - Update L: $[L \mapsto (A : a + a', B : b + b')]_{A,B}$
4. Now the sub-channel χ has been defunded, it can be safely discarded.

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, χ:
   - $[\![L{:}x]\!],\ [L \mapsto (A : a, B : b, \chi : c)]_{A,B}$, where $x = a + b + c$.
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.
   - Update channel L: $[L \mapsto (B : b, \chi : c, A : a + a')]_{A,B}$
3. It is now safe for A to deposit into the channel on-chain:
   - Deposit by A: $D_L(a')[\![L{:}x]\!] = [\![L{:}x + a']\!]$
4. Finally, if required, the participants can reorder the state again:
   - Update channel L: $[L \mapsto (A : a + a', B : b, \chi : c)]_{A,B}$

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:
   - $[\![L{:}x]\!],\ [L \mapsto (A : a + a', B : b, \chi : c)]_{A,B}$, where $x = a + a' + b + c$.
2. The participants start by creating 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.
   - Create channel L: $[L' \mapsto (A : a, B : b, \chi : c)]_{A,B}$
3. They then update L to fund L' alongside the coins that A wants to withdraw. They conclude the channel in this state:
   - Update channel L: $[L \mapsto (L' : a + b + c, A : a')]_{A,B}$
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:

- Finalize L on-chain: $[\![L{:}x \mapsto (L' : a+b+c, A : a')]\!]$

5. A can then call the transfer operation to get their coins under their control. They can optionally move the coins into L' at the same time:
   - Transfer coins to A: $T_{L,A}(a')[\![L{:}x \mapsto (L : a+b+c, A : a')]\!] = [\![L{:}x - a' \mapsto (L' : a+b+c),\ A{:}a]\!]$
   - Transfer coins to L': $T_{L,L'}(a+b+c)[\![L{:}x \mapsto (L : a+b+c),\ A{:}a']\!] = [\![L'{:}a+b+c,\ A{:}a]\!]$

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

5 Nitro

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.

We will use the notation $(B|C, D)$ for a guarantee with target B, which prioritizes first C, then D, then to any other addresses according to the priorities in B's allocation. We say a guarantee channel, A, which targets an allocation channel, B, 'can afford x for C', if C would receive at least x coins, were the coins currently held in A to be paid out according to A's reprioritization of B's allocation.

Nitro adds the **claim** operation, $C_{A,C}(x)$, to the existing transfer, deposit and withdraw operations. If A acts as guarantor for B and can afford x for C, then $C_{A,C}(x)$: (a) reduces the funds held by A by x, (b) increases the funds held by C by x, and (c) reduces the amount owed to C in the outcome of B. If the outcome of B is not yet registered on-chain, or if A cannot afford x for C, then the operation has no effect.

Example 5.1. In the following example, we have a guarantee channel, G, which holds 5 coins and guarantees L's allocation, with χ as highest priority.

$$C_{G,\chi}(2)[\![G{:}5 \mapsto (L|A), L{:}(A:5, \chi:5)]\!] = [\![G{:}3 \mapsto (L|A), L{:}(A:5, \chi:3), \chi{:}4]\!]$$

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

We give a python implementation of the Nitro adjudicator 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 will now give the construction for the simplest possible virtual channel, between A and B through a shared intermediary, C. Our starting point for this channel is a pair of ledger channels, L and L', with participants $\{A, C\}$ and $\{B, C\}$ respectively.

$$[\![L{:}x, L'{:}x]\!],\ [L \mapsto (A : a, C : b)]_{A,C},\ [L' \mapsto (B : b, C : a)]_{B,C} \tag{1}$$

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 $\{A, B, C\}$ and two guarantor channels G and G' which target J. The setup is shown in figure 1.

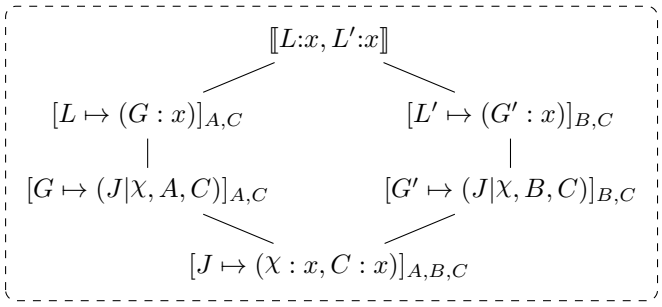

Figure 1: Virtual channel construction

We will cover the steps for safely setting up this construction in section 5.2. In the rest of this section we will explain why this construction can be considered to fund the channel χ. Similarly to the method for ledger channel construction, we will do this by demonstrating how any one of the participants can offload the channel χ: converting it to an on-chain channel that holds its own funds.

We will first consider the case where A wishes to offload χ. We will assume that A starts by finalizing all their finalizable channels on-chain, followed by a transferring funds held in the ledger channel L to the guarantor channel G. The final step is for A to claim on G's guarantee for χ:

$$C_{G,\chi}(x)[\![G{:}x \mapsto (J|\chi, A, C), L'{:}x, J \mapsto (\chi : x, C : x)]\!] = \\ [\![L'{:}x, \chi{:}x, J \mapsto (C : x)]\!] \tag{2}$$

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 C wants to offload the channel and reclaim their funds. This is important to ensure that A and B cannot lock C's

funds indefinitely in the channel. We assume that C has started by registering all their finalizable channels on-chain, followed by transferring funds from L to G and from L' to G'.

$$C_{G',G}(C)C_{G,\chi}(x)[\![G{:}x \mapsto (J|\chi, A, C), G'{:}x \mapsto (J|\chi, B, C), \\ J \mapsto (\chi : x, C : x)]\!] = [\![L'{:}x, \chi{:}x, C{:}x]\!] \quad (3)$$

Note that C has to claim on both guarantees, offloading χ before being able to reclaim their funds.

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.

5.2 Opening and Closing Virtual Channels

In this section we present a sequence of system 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 system 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 (1):
   - $[\![L{:}x, L'{:}x]\!]$, $[L \mapsto (A : a, C : b)]_{A,C}$, $[L' \mapsto (B : b, C : a)]_{B,C}$
2. A and B bring their channel χ to the funding point:
   - Create channel χ: $[\chi \mapsto (A : a, B : b)]_{A,B}$
3. In any order, A, B and C setup the virtual channel construction:
   - Create channel J: $[J \mapsto (A : a, B : b, C : c)]_{A,B,C}$
   - Create channel G: $[G \mapsto (J|\chi, A, C)]_{A,C}$
   - Create channel G': $[G' \mapsto (J|\chi, B, C)]_{B,C}$
4. In either order switch the ledger channels over to fund the guarantees:
   - Update L: $[L \mapsto (G : x)]_{A,C}$
   - Update L': $[L' \mapsto (G' : x)]_{B,C}$
5. Switch J over to fund χ:
   - Update channel J: $[J \mapsto (\chi : x, C : x)]_{A,B,C}$

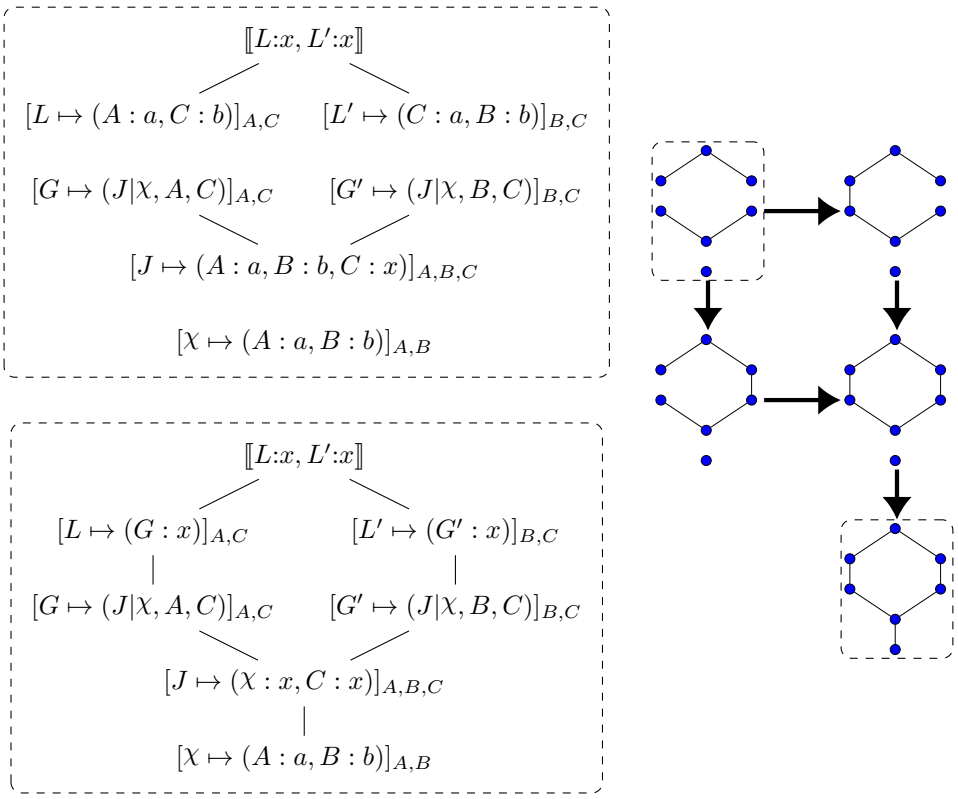

Figure 2: Opening a virtual channel

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 χ by signing a conclusion proof:
   - Update χ: $[\chi \mapsto (A : a', B : b')]_{A,B}$
2. A and B sign an update to J to take account of the outcome of χ. C will accept this update, provided that their allocation of x coins remains the same:
   - Update channel J: $[J \mapsto (A : a', B : b', C : x)]_{A,B,C}$
3. In either order switch the ledger channels to absorb the outcome of J, defunding the guarantor channels in the process:
   - Update L: $[L \mapsto (A : a', C : b')]_{A,C}$

- Update L': $[L' \mapsto (B : b', C : a')]_{B,C}$

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

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

6 Acknowledgements

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

7 Appendix

7.1 Overview of ForceMove

7.2 The Consensus Game

7.3 Virtual Channels on Turbo

7.4 Payouts to Non-Participants

7.5 Possible Extensions

7.6 On-chain Operations Code

7.6.1 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
```

7.6.2 Remove

```
def remove(outcome, recipient, amount):
  newOutcome = []
  for [address, allocation] in outcome:
    if address == recipient:
      reduction = min(allocation, amount)
      amount = amount - reduction
      newOutcome.append([address, allocation - reduction])
    else:
      newOutcome.append([address, allocation])

  return newOutcome
```

7.6.3 Transfer

```
from overlap import overlap
from remove import remove

def transfer(recipient, outcome, funding, amount):
  if overlap(recipient, outcome, funding) >= amount:
    funding = funding - amount
    outcome = remove(outcome, recipient, amount)
  else:
    amount = 0

  return (amount, outcome, funding)
```

7.6.4 Claim

```
from overlap import overlap
from remove import remove
from cap import cap

def claim(recipient, guarantee, outcome, funding, amount):
  cappedGuarantee = cap(guarantee, outcome)
  if overlap(recipient, cappedGuarantee, funding) >= amount:
    funding = funding - amount
    guarantee = remove(cappedGuarantee, recipient, amount)
    outcome = remove(outcome, recipient, amount)
  else:
    amount = 0

  return (amount, guarantee, outcome, funding)
```