Nitro Protocol

Tom Close

January 7, 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

In this paper, we are focus on the construction of networks of state channels, which we treat separately from the operation of those channels. For the purpose of this work, we view a state channel and its states purely as a device for enabling a given party to realize a given outcome on-chain, in a sense that we will make precise. In particular, we do not specify the state format, the update rules, the mechanics of on-chain challenges or how to respond to them.

The ForceMove framework is an example of a state channel framework that specifies all these details. The state channel networks presented here were designed to work with ForceMove and we will frequently present examples involving ForceMove channels. That said, the techniques provided here should be applicable to any state channel framework sharing the features presented in this section.

We start the section by reviewing the properties of ForceMove that are important for Turbo and Nitro [1]. We then introduce some concepts that will be useful later when reasoning about the correctness of our state channel networks.

3.1 ForceMove Channels

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'.

3.2 Depositing, Holding and Withdrawing Coins

In order to have value, a state channel system must be backed by assets held onchain. 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.

[1] We give a more complete recap of ForceMove in the appendix.

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 $[\![\alpha_\chi(x)]\!]$, where the α denotes funds being held and the $[\![\quad]\!]$ is used to indicate 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.

$$D_\chi(x)[\![\alpha_\chi(y)]\!] = [\![\alpha_\chi(x+y)]\!]$$

In order to distribute the coins it holds, a state channel must have one or more mechanisms for registering its **outcome**, ω, 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 particpants indicating that the channel has concluded, thus allowing them to skip the challenge timeout. We write $[\![\beta_\chi(\omega)]\!]$ to represent the situation where the outcome ω has been registered on-chain for channel χ.

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:

$$O[\![\alpha_\chi(a+b)\beta_\chi(\omega)]\!] = [\![\alpha_A(a)\alpha_\chi(b)\beta_\chi(\omega')]\!]$$

The specification of the precise format for the outcome, ω, 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

$$W_A(x)[\![\alpha_A(x')]\!] = [\![\alpha_A(x'-x)]\!]$$

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 [2].

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

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 β_χ; the intermediate state, α_A, where funds are held against a participant address never exists on-chain.

3.3 The Value of a State

The utility of a state channel comes from the ability of being able 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 withdraw 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 withdraw x coins.

In terms of applying operations on-chain, we need to If it is currently time t, and I want to apply operation O on-chain before time $t + \epsilon$ then it is possible for me to do so. But I cannot prevent you from applying O' beforehand. Ignore gas + censorship, allow front-running

In general, as a state channel system transitions from one state to another the value of the system for each participant will change - otherwise state channels would be useless!

- focus of this paper is the creation of state channel networks - frequently have to transform the system state - it will be important that these are value preserving

- principle of value equivalence - if two states are value equivalent then the participants will do them

- also slight modification: for deposits

3.4 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 $[\beta_{\mathcal{X}}(\omega)]_A$, to represent a state of a channel, $\mathcal{X}$, where the outcome, ω, is finalizable by A.

$$[\beta_{\mathcal{X}}(\omega)]_A \xrightarrow{\text{A's unbeatable strategy}} [\![\beta_{\mathcal{X}}(\omega)]\!]$$

It follows from the definition that exactly one of the following statements is true about a channel $\mathcal{X}$ at any point in time:

1. No participants apart from p have a finalizable outcome. Participant p has one or more finalizable outcome(s), $\omega_1, \ldots, \omega_m$. We write this $[\beta_{\mathcal{X}}(\omega_1) \ldots \beta_{\mathcal{X}}(\omega_m)]_p$.
2. There are at least two participants, $P = \{p_1, \ldots, p_m\}$, who share the same finalizable outcome, ω. We write this $[\beta_{\mathcal{X}}(\omega)]_{p_1,\ldots,p_m}$.
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. 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 ω, then ω 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.

[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 $[\beta_1 \dots \beta_m]_{(p)}$.

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, from changing. At the midpoint, when state σ_n has just been broadcast, the outcome ω is finalizable for every participant, i.e. we have $[\beta_\chi(\omega)]_{p_1,\dots,p_n}$. The outcome ω 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, that is finalizable for every participant, i.e. $[\beta_\chi(\omega)]_{p_1,\dots,p_n}$. We will sometimes write this state using the shortened notation $[\beta_\chi(\omega)]_P$.

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.

3.5 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, ω_1, to another, ω_2. In order for this to happen, one participant proposes the new outcome, ω_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 ω_1 and ω_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.

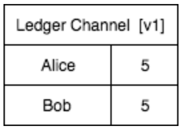
Value Preserving Consensus Transitions (VPCTs): If I have two system states, Σ and Σ', which only differ in the state of a consensus game chanel, χ, with participants P and universally enforceable outcomes $[\beta_\chi(\omega)]_P \in \Sigma$ and $[\beta_\chi(\omega')]_P \in \Sigma'$, and for some participant $p \in P$, $\nu_p(\Sigma) = \nu_p(\Sigma') = x$, 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 or ω' on-chain, then they can obtain value x. The consensus transition allows the participants to transition from outcome ω to ω' 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 Protocol

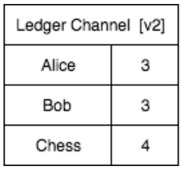
Turbo protocol allows a set of participants who already have a funded channel to open and close sub-channels without any on-chain transactions. In order to do this, the protocol specifies the format of the channel outcomes and how they are interpreted on-chain. It does not specify the channel update and challenge mechanics but can be combined with ForceMove to provide a fully functional system.

As an example of what Turbo enables, suppose Alice and Bob want to play a game of chess and that they already have an existing state channel, χ_L. The winner of the game of chess should receive 2 coins from the loser and χ_L currently holds 5 of Alice's coins and 5 of Bob's.

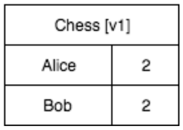


Ledger Channel [v1]	
Alice	5
Bob	5

Alice and Bob proceed by creating a new channel χ_C for the chess game with the appropriate starting state. They then update χ_L to allocate funds to these games.

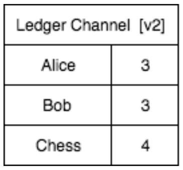


Ledger Channel [v2]	
Alice	3
Bob	3
Chess	4

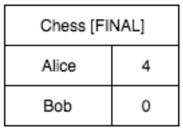


Chess [v1]	
Alice	2
Bob	2

Once the funds are allocated, they are free to play the game of chess. Updates to the chess channel are independent from updates to χ_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.

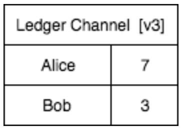


Ledger Channel [v2]	
Alice	3
Bob	3
Chess	4



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

In the rest of this section we will present the protocol that makes the above interaction possible.

4.1 Turbo Outcomes and Operations

Turbo protocol specifies the interpretation of channel outcomes and the operations that manipulate them once they are on-chain. The process of registering the outcome with the chain is not specified, but could be accomplished using the ForceMove protocol.

Outcomes in Turbo take the form of an ordered list of address-value pairs. To write outcomes we will use the notation $\{a_1{:}v_1, \ldots, a_n{:}v_n\}$, where the a_i represent addresses and the v_i represent values. The key idea behind Turbo is that the addresses in the outcomes are not limited to being participant addresses but that they can also be addresses of other *channels*.

The ordering in the outcome is significant and is used to determine a priority order for payouts, which becomes important if channel does not hold enough funds to cover the entire outcome.

The naive algorithm for distributing the funds is to work along the outcome from the front, paying funds out to the corresponding address until no funds are remaining. In practice, we avoid introducing a dependency on the order of the payouts by using the **overlap** function `overlap`(p, ω, x), which returns the payout owed to p from outcome ω when the channel is funded with x. If we have $\llbracket \beta_\chi(\omega) \rrbracket$ and `overlap`$(p, \omega, x) = y$, we say that χ **allocates** y to p **within** x. A python implementation of `overlap` is given in the appendix.

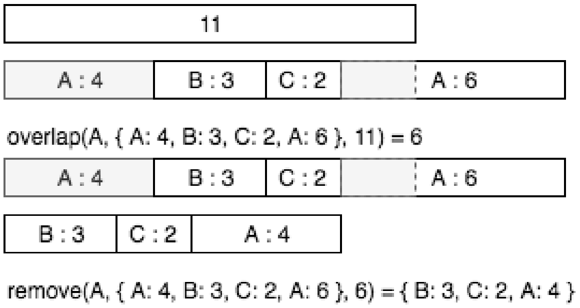

11
A : 4
B : 3
C : 2
A : 6
overlap(A, { A: 4, B: 3, C: 2, A: 6 }, 11) = 6
A : 4
B : 3
C : 2
A : 6
B : 3
C : 2
A : 4
remove(A, { A: 4, B: 3, C: 2, A: 6 }, 6) = { B: 3, C: 2, A: 4 }

Figure 1: Overlap and remove

Once an outcome, ω, has paid out a total, x, to an address, A, we will **remove** the total to create a new outcome $\omega' =$ `remove`(A, ω, x). The remove operation works through the outcome from the start, reducing the totals for A until x coins have been removed. A python implementation of `remove` is given in the appendix.

Armed with overlap and remove, we can now define the only on-chain operation added by Turbo. The **transfer** operation, $T_{A,B}(x)$, is an instruction to transfer x coins currently held by A to B, according to the outcome of A.

The operation can be described as follows: if channel A holds at least $a \geq x$ coins and the outcome, $\beta_A(\omega)$, of A allocates x to B within a, then decrease the total held by A by x coins, increase the total held by B by x coins and replace the outcome with $\beta_A(\omega')$, where $\omega' =$ `remove`(B, ω, x). Otherwise, do nothing. We provide a python implementation of transfer in the appendix. By way of an example:

$$T_{A,B}(3)\llbracket \alpha_A(10)\beta_A(B:3, C:7) \rrbracket = \llbracket \alpha_A(7)\alpha_B(3)\beta_A(C:7) \rrbracket$$

Combined with the deposit and withdrawal operations, the transfer operation gives us everything we need to create a system that supports ledger channels.

4.2 Value Equivalence in Turbo

In section 3.3 we defined the value of a system state, $\nu_p(\Sigma)$, to be the to be the largest x such that p has an unbeatable strategy to extract x coins from the adjudicator. We then split this into two parts: the unbeatable strategy for registering an outcome with the adjudicator and the unbeatable strategy for calling on-chain operations to move the coins into p's address. We looked a lot at the former part, which gave us the concept of a finalizable state, but have not yet covered the second part. In this section, we will look at what it takes to have an unbeatable strategy for moving funds from one address to another in Turbo.

For Turbo it turns out that the rule is very simple: if p has any strategy for withdrawing x coins from the adjudicator, it is an unbeatable strategy.

The following observation about the transfer operation is the key portion of the proof: if I have the operation $T_{A,B}(x)$, that causes a change to the on-chain state, then we must have some outcome $[\![\beta_A(\omega)]\!]$, which means that A is a channel address and therefore withdrawals from A are impossible. Therefore the only operations that can reduce the total held in A are transfers of the form $T_{A,\chi}(x)$ for addresses χ that appear in ω. Due to the definition of the transfer operation, all operations of this form commute. So no transfer operation inserted in front of $T_{A,\chi}(x)$ can change its effect on the state.

4.3 Ledger channels

While this generalization allows for a variety of relationships between different channels, we will focus here on a setup where a single parent channel funds one or more sub-channels [5]. Following the Perun paper, we will refer to this parent channel as a **ledger channel**.

All ledger channels in the following sections run the consensus game, so that we can use the value preserving consensus transition result.

4.3.1 Opening a sub-channel

In this section, we give the details of the sub-channel opening process that we touched on at the beginning of the Turbo chapter. We assume we start in a position where we have a ledger channel L between A and B, which contains

[5] Note that we allow the case where the sub-channels are themselves ledger channels.

enough to fund a sub-channel, $\mathcal{X}$. This sub-channel runs an arbitrary ForceMove application and starts the process at the funding point, giving the universally finalizable below. We progress through the following system states:

$$\begin{aligned}
S_1 &= \llbracket \alpha_L(a+b) \rrbracket \quad [\beta_L(A:a, B:b)]_{A,B} \\
S_2 &= \llbracket \alpha_L(a+b) \rrbracket \quad [\beta_L(A:a, B:b)]_{A,B} \quad [\beta_{\mathcal{X}}(A:a', B:b')]_{A,B} \\
S_3 &= \llbracket \alpha_L(a+b) \rrbracket \quad [\beta_L(A:a-a', B:b-b', \mathcal{X}:x)]_{A,B} \quad [\beta_{\mathcal{X}}(A:a', B:b')]_{A,B}
\end{aligned}$$

where $x = a' + b'$. For all of these states, we have $\nu_A(S_i) = a$ and $\nu_B(S_i) = b$. As each system state only differs in the state of one channel, we can apply the VPCT result, to see that we can move between the system states whilst keeping the system value constant for both A and B. At the end of this process, the funding of the sub-channel is complete and the players can proceed with the non-value preserving interaction within it.

4.3.2 Closing a sub-channel

Closing a sub-channel is very similar to opening a sub-channel, involving the same states in the reverse order. In this instance, the universally finalizable state in the sub-channel, $\mathcal{X}$, is taken to come from a conclusion proof.

$$\begin{aligned}
S_1 &= \llbracket \alpha_L(x) \rrbracket \quad [\beta_L(A:a, B:b, \mathcal{X}:a'+b')]_{A,B} \quad [\beta_{\mathcal{X}}(A:a', B:b')]_{A,B} \\
S_2 &= \llbracket \alpha_L(x) \rrbracket \quad [\beta_L(A:a+a', B:b+b')]_{A,B} \quad [\beta_{\mathcal{X}}(A:a', B:b')]_{A,B} \\
S_3 &= \llbracket \alpha_L(x) \rrbracket \quad [\beta_L(A:a+a', B:b+b')]_{A,B}
\end{aligned}$$

where $x = a+a'+b+b'$. The analysis here is exactly the same as in the opening case, where $\nu_A(S_i) = a + a'$ and $\nu_B(S_i) = b + b'$. We then apply the VPCT result to argue that we can transition between these states without changing the value for either player.

4.3.3 Topping up a ledger channel

Sometimes it is useful for one participant to be able to increase their total in a ledger channel without disturbing any sub-channels operating in it. In the scenario below, A and B have a ledger channel, L, supporting a sub-channel, $\mathcal{X}$, whose state is not shown. Participant A wants to increase their balance in the channel by an amount a', through an on-chain deposit. The operation proceeds as follows:

$$\begin{aligned}
S_1 &= \llbracket \alpha_L(x) \rrbracket & & [\beta(A:a, B:b, \mathcal{X}:c)]_{A,B} \\
S_2 &= \llbracket \alpha_L(x) \rrbracket & & [\beta(B:b, \mathcal{X}:c, A:a+a')]_{A,B} \\
S_3 &= D_L(a')\llbracket \alpha_L(x) \rrbracket & & [\beta(B:b, \mathcal{X}:c, A:a+a')]_{A,B} \\
S_4 &= \llbracket \alpha_L(x+a') \rrbracket & & [\beta(B:b, \mathcal{X}:c, A:a+a')]_{A,B}
\end{aligned}$$

Here we have $\nu_A(S_1) = \nu_A(S_2) = a + c_A$, $\nu_A(S_3) = \nu_A(S_4) = a + a' + c_A$ and the value to B stays constant at $b + c_B$ throughout, where c_A and c_B is the value attributable to A and B, respectively, from the $\mathcal{X}$ sub-channel. The operation starts by moving A's entry to the end of the list. It is then safe to increase the total to $a + a'$, using the priority order payout property and the fact that the channel is underfunded, to see that this has not changed the value to A. In the next line, A deposits a', which causes the value to A to increase by the same amount, as should happen. At this point the channel is fully funded and the participants are free to reorder the entries of the outcome as they please.

4.3.4 Partial checkout from a ledger channel

A partial checkout is the opposite case to a top up: here A wants to withdraw a' coins from the ledger channel, without disturbing any sub-channels. The approach here is different to topping up. The general idea is to replace the ledger channel, L, with an equivalent ledger channel, L', with A's balance reduced and then switch the funding across. For simplicity, we assume that we start with $\llbracket \alpha_A(0) \rrbracket$, so A has no funds held on-chain at the beginning.

$$
\begin{aligned}
S_1 &= \llbracket \alpha_L(x) \rrbracket \quad [\beta_L(B : b, A : a, \mathcal{X} : c)]_{A,B} \\
S_2 &= \llbracket \alpha_L(x) \rrbracket \quad [\beta_L(B : b, A : a, \mathcal{X} : c)]_{A,B} \quad [\beta_{L'}(B : b, A : a - a', \mathcal{X} : c)]_{A,B} \\
S_3 &= \llbracket \alpha_L(x) \rrbracket \quad [\beta_L(L' : x - a', A : a')]_{A,B} \quad [\beta_{L'}(B : b, A : a - a', \mathcal{X} : c)]_{A,B} \\
S_4 &= \llbracket \alpha_L(x) \beta_L(L' : x - a, A : a') \rrbracket \quad [\beta_{L'}(B : b, A : a - a', \mathcal{X} : c)]_{A,B} \\
S_5 &= \llbracket \alpha_{L'}(x - a') \alpha_A(a') \rrbracket \quad [\beta_{L'}(B : b, A : a - a', \mathcal{X} : c)]_{A,B}
\end{aligned}
$$

All these states have the same value for A and B. The first two transitions rely on the VPCT result. The move from S_3 to S_4 involves registering the outcome of L on-chain. It is assumed here that the participants create a conclusion proof, so that they can skip the timeout. The final transition involves applying the transfer operations $T_{L,L'}(x - a')$ and $T_{L,A}(a')$.

5 Nitro Protocol

Nitro Protocol is an extension to Turbo that introduces a new type of outcome and the on-chain operations to support it. We will show how this extension enables a channel between two participants to be supported through a third party. Following the terminology in the Perun paper, we will refer to these channels as **virtual channels**.

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 Carol and Bob and Carol have a $\{B : 5, C : 5\}$ ledger channel open too.

Nitro Protocol allows Alice and Bob to open their chess game, using their existing ledger channels with Carol, 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 $\{A : 10, B : 0\}$. Then Nitro allows Alice and Bob to close their chess game off-chain too, rebalancing through Carol, so that the final state of the ledger channels is $\{A : 10, C : 0\}$ and $\{B : 0, C : 10\}$.

In this interaction, Carol contributed 10 of her coins at the beginning, which had to remain locked while Alice and Bob were playing. She got those 10 coins back 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.

In this section, we will introduce the new outcomes and operations, along with the network construction to make interactions like the above possible.

5.1 Nitro Outcomes and Operations

Nitro extends Turbo by adding a new type of channel whose outcomes are interpreted differently. We refer to these new channels as **guarantor** channels. In all ways apart from the interpretation of the outcomes, guarantor channels behave exactly like regular channels: they follow the ForceMove update rules and are finalized on-chain in the same way, and they can run ForceMove applications, like the consensus game.

The outcome of a guarantor channel specifies certain parts of another channel's outcome to guarantee. We will use the notation $\gamma_{A,B}(\omega)$ to represent the outcome, ω, of channel A, which is acting as a guarantor for B.

Along with the new type of channel, Nitro adds one operation: the **claim** operation, $G_{A,B,C}(x)$, which is used to claim on the guarantee of A on B's payout to C.

Before proceeding with the definitions, we are going to start with some examples to give an idea of how these new additions interact:

Example 5.1. This example shows the simplest case of claiming on a guarantee:

$$G_{A,B,C}(5)[\![\alpha_A(5)\gamma_{A,B}(C:5)\beta_B(C:5)]\!] = [\![\alpha_C(5)]\!]$$

Guarantor channel A holds 5 and guarantees 5 for Bs allocation to C, within 5. B allocates 5 to C, so the claim operation is successful, paying 5 from A to C through the outcome of B.

Example 5.2. The next example shows a slightly more complicated case:

$$G_{A,B,C}(2)[\![\alpha_A(3)\gamma_{A,B}(C:2)\beta_B(C:3)]\!] = [\![\alpha_C(2)\alpha_A(1)\beta_B(C:1)]\!]$$

Here 2 is transferred from A to C through B, but this does not completely eliminate B's payout to C.

Example 5.3. The guarantee only goes through if the balances in α, γ and β are all large enough.

$$G_{A,B,C}(3)[\![\alpha_A(3)\gamma_{A,B}(C:2)\beta_B(C:3)]\!] = [\![\alpha_A(3)\gamma_{A,B}(C:2)\beta_B(C:3)]\!]$$

The operation has no effect, as the balance in the guarantee is not high enough.

Example 5.4. The guarantee can be used to modify the order of payouts:

$$G_{A,B,C}(2)[\![\alpha_A(2)\gamma_{A,B}(C:2,D:1)\beta_B(D:1,C:2)]\!] = \\ [\![\alpha_C(2)\gamma_{A,B}(D:1)\beta_B(D:1)]\!]$$

Note that in this case, the payout to C went through because it appeared first in the guarantee outcome, even though it was lower priority in B's outcome.

Example 5.5. Multiple guarantees can also be used to cover the outcome of a single channel:

$$G_{A',B,D}(2)G_{A,B,C}(2) \\ [\![\alpha_A(2)\alpha_{A'}(2)\gamma_{A,B}(C:2,D:2)\gamma_{A',B}(C:2,D:2)\beta_B(C:2,D:2)]\!] \\ = [\![\alpha_C(2)\alpha_D(2)]\!]$$

In this example, we managed to set up a situation where it did not matter which of A and A' went to C and which went to D but C would always receive their funds first. This trick will be used in the virtual channel construction.

In order to specify the claim operation, it will be useful to introduce a utility operation first: the **cap** operation. By capping outcome ω_1 by ω_2 you ensure that ω_1 will never pay out more to any address A than ω_2 would. The python code for the cap operation is given in the appendix (7.6.4), so here we will settle for a few examples:

$$\begin{aligned} \texttt{cap}(\{A:10\},\{A:4\}) &= \{A:4\} \\ \texttt{cap}(\{A:10\},\{A:4,B:2,A:4\}) &= \{A:8\} \\ \texttt{cap}(\{A:2,B:3,A:3,D:2\},\{A:1,B:2,A:2\}) &= \{A:2,B:2,A:1\} \end{aligned}$$

We will use the cap operation to define the **claim** operation, $T_{A,B,C}(x)$, as follows: if A holds a, and the outcome of A capped by the outcome of B allocates x to C within y, then decrease the amount held by A by x, increase the amount held by C by x, remove x from A's allocation to C and also remove x from B's allocation to C. A python implementation of the claim operation can be found in 7.6.5.

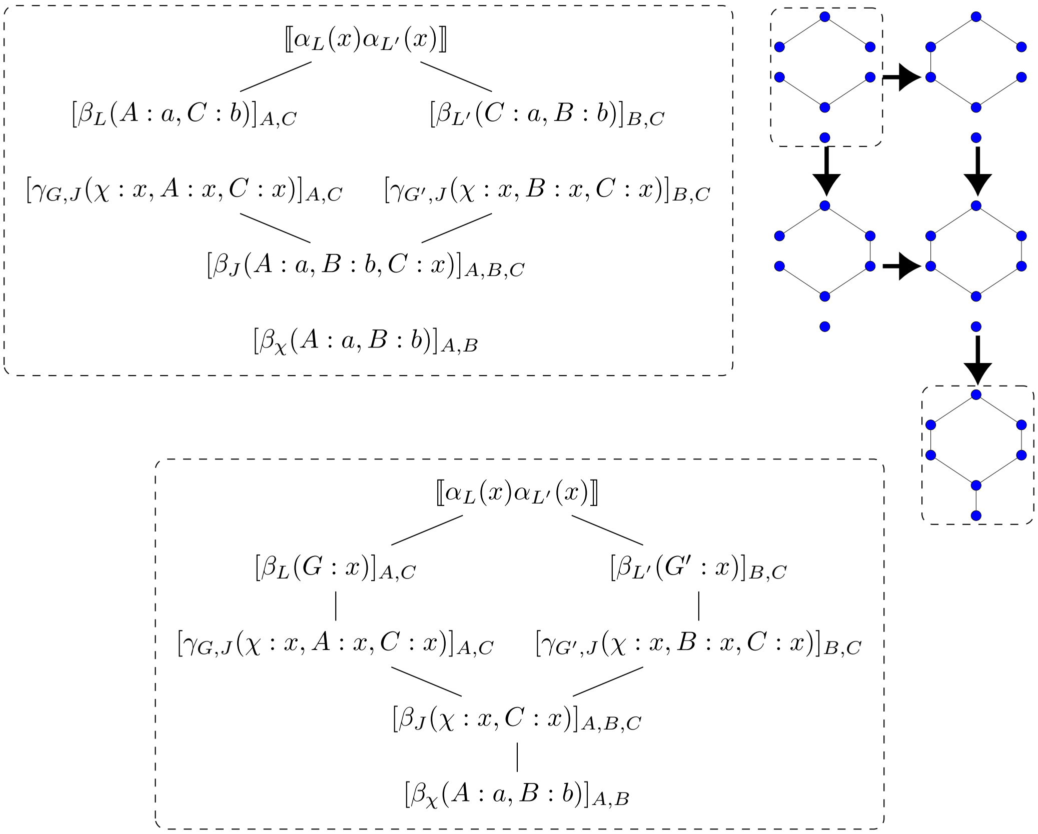
5.2 Value Calculations

$$T_{B,D}(5)G_{A,B,C}(5)[\![\alpha_A(5)\alpha_B(5)\gamma_{A,B}(C:5)\beta_B(C:5,D:5)]\!] \\ = [\![\alpha_C(5)\alpha_D(5)]\!]$$

$$T_{B,C}(5)[\![\alpha_A(5)\alpha_B(5)\gamma_{A,B}(C:5)\beta_B(C:5,D:5)]\!]$$
$$= [\![\alpha_C(5)\alpha_A(5)\gamma_{A,B}(C:5)\beta_B(D:5)]\!]$$

5.3 Virtual Channels

- outline how we can use guarantee channels to create virtual channels - the construction we will use is as follows



$[\![\alpha_L(x)\alpha_{L'}(x)]\!]$
$[\beta_L(A:a,C:b)]_{A,C}$
$[\beta_{L'}(C:a,B:b)]_{B,C}$
$[\gamma_{G,J}(\chi:x,A:x,C:x)]_{A,C}$
$[\gamma_{G',J}(\chi:x,B:x,C:x)]_{B,C}$
$[\beta_J(A:a,B:b,C:x)]_{A,B,C}$
$[\beta_\chi(A:a,B:b)]_{A,B}$
$[\![\alpha_L(x)\alpha_{L'}(x)]\!]$
$[\beta_L(G:x)]_{A,C}$
$[\beta_{L'}(G':x)]_{B,C}$
$[\gamma_{G,J}(\chi:x,A:x,C:x)]_{A,C}$
$[\gamma_{G',J}(\chi:x,B:x,C:x)]_{B,C}$
$[\beta_J(\chi:x,C:x)]_{A,B,C}$
$[\beta_\chi(A:a,B:b)]_{A,B}$

Figure 2: Ledger channels, $x = a + b$

- worth thinking about how this works - offload - key is the joint A-B-C channel, source of truth for state of channel

5.3.1 Opening a Virtual Channel

Want to start in the situation where channels A-C and B-C exist and demonstrate how to open the virtual channel situation

- first create the subchannel we want to fund - and the joint channel and guarantee channels to support it everything here is disconnected - then we update the ledger channels, one at a time to fund the guarantees - finally we update the joint channel to fund the virtual channel

5.3.2 Closing a Virtual Channel

Opening a channel in reverse - start with the virtual channel coming to a finalizable outcome e.g. conclude - then A or B propose an update to j - once A and B have signed off, it's in C's interest too - then they can defund the guarantee channels independently

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):
  result = 0
  for [address, allocation] in outcome:
    if funding <= 0:
      break;

    if address == recipient:
      result += min(allocation, funding)
    funding -= allocation

  return result
```

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 Cap

```
from collections import defaultdict

def cap(outcome, outcome2):
  totals = defaultdict(int)
  for [address, allocation] in outcome2:
    totals[address] += allocation

  cappedOutcome = []
  for [address, allocation] in outcome:
    remaining = totals[address]
    newAllocation = min(allocation, remaining)
    cappedOutcome.append([address, newAllocation])
    totals[address] -= newAllocation

  return cappedOutcome
```

7.6.5 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)
```