Nitro Protocol

Tom Close

January 4, 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

common features of state channels - participants + signatures

- addresses - forceMove ordered set of participants + nonce - no collisions between different sets of participants

- assets held on chain, how do we record values - values

- deposits and withdrawals

- ignore states + updates mechanism, challenge mechanism - outcomes $\beta_{\chi}(\omega)$

The **participants** of a state channel are defined by a cryptographic address, co - all addresses generated from the signature scheme - someone somewhere should know the private key corresponding to a participant address - they are the owner of that address - a participants can own multiple addresses - the key is used to sign updates - addresses don't have to have any funds allocated on-chain - they can be ephemeral

- space of channels - ordered subset of participants - a nonce - channel address - non-collisions

In this paper we will focus on state channels that determine the distribution of a single asset, which we will refer to as 'coins'.

- in this paper we deal with a single asset which we will refer to as coins - the assets allocated can be summarised with a single integer - in this paper we allow the coin value to take any value in - this is not realistic for the blockchain context, where there is typically a maximum value, `MAX`, after which totals overflow - meaning that we should really consider coins in the space $\mathbb{Z}_{\texttt{M}}$. - we can also consider state channels that operate on multiple assets: $\mathbb{Z}_{\texttt{M}} \times \mathbb{Z}_{\texttt{M}}$

- space of application rules. Consider these to also define the states - state transitions - payment channel can be thought of in terms of a state channel system with a single application - is it possible to add to the space of application rules - what sort of rules does it support? transition rules depend on time or the general state of the chain - we assume here that they don't - possible to remove these restrictions

- space of channel addresses - participant addresses - channels = set of participants + nonce - channel addresses

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

3.2 Funding and Channel Outcomes

- adjudicator: two important collections - funds held and outcomes - holding funds + notation - holding outcomes

- outcomes of a channel - outcome: how funds should be distributed - notation - make this precise later

- ways of registering outcomes in force move - only one outcome on-chain for each channel. Once one outcome has been registered no other one can be - abstract a channel to be something that can reach an outcome

- first part of this is what outcomes you can register on-chain

- replace the simpleAdjudicator - do we need alpha and deposits

3.3 Finalizable and Enabled Outcomes

- state of a channel - outcome is finalizable if you can register it on-chain and no-one can stop you. - an unbeatable strategy for getting it on-chain - series of actions

Example 3.1. The next mover in ForceMove. In a ForceMove state channel with n participants, if participant p has just received state σ_m with $m > n$ and it is their turn to sign state $m + 1$, then we have $[\sigma_m.\beta]_p$ - the state's balances $\sigma_m.\beta$ is a finalizable outcome for p. Why? To finalize this outcome p can call a force-move on σ_m and then fail to respond within the timeout. [The reason why ForceMove always allows a transition to a conclude state is to give p a way of accomplishing the same outcome off-chain.]

Example 3.2. Two conclusion proofs. In a ForceMove state channel, if two different conclusion proofs with different outcomes, $\beta_1 \neq \beta_2$, are both held by participants p_1 and p_2, then no outcome is finalizable by any participant. Why? No finalizable outcome other than β_1 is possible any party p, as it is always possible for either p_1 or p_2 to register the conclusion proof resulting β_1 immediately. But the same is also true for β_2. Therefore no finalizable outcome exists. [For this reason, participants should never sign more than one conclusion state.]

- enabled outcomes

Example 3.3. Pre-fund setup in ForceMove. In a ForceMove state channel with n participants P, at end of the pre-fund setup i.e. when the last state to be broadcast was σ_n, then we have $[\sigma_n.\beta]_P$ - the state's balances $\sigma_n.\beta$ is finalizable for all participants of the channel. Why? According to the ForceMove transition rules, the outcome must remain unchanged for the first $2n$ states. That means that every participant must contribute another signature before the outcome of the state can change. By with-holding this signature and forcing the channel to conclusion through a series of force-moves, any participant can ensure that the current outcome is registered on-chain.

3.4 Calculating Value

- value (unbeatable strategy for extracting at least a given total, without assuming we add funds) - value equivalent (if two system states have the same value) - value preserving transition - applications change value, opening and closing an application shouldn't

- transitioning between outcomes - rewrite rules - channels update independently

- possible to transition between finalizable states

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.

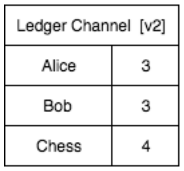
Ledger Channel [v1]	
Alice	5
Bob	5

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.

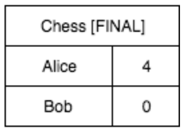
Ledger Channel [v2]	
Alice	3
Bob	3
Chess	4

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.

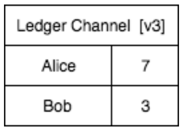


Ledger Channel [v2]	
Alice	3
Bob	3
Chess	4



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.

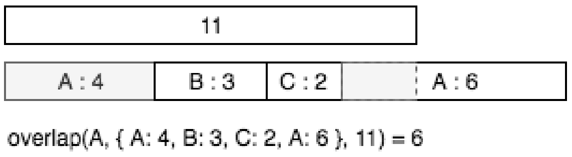
Outcomes in Turbo take the form of an ordered list of address-value pairs. To write outcomes we will use the notation $\{a_1{:}v_1, \ldots, a_n{:}v_n\}$, where the a_i represent addresses and the v_i represent values. The key idea behind Turbo is that the addresses in the outcomes are not limited to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

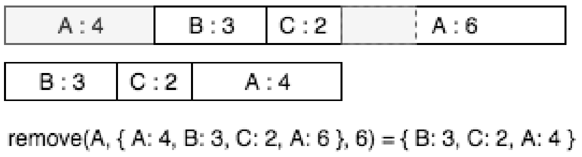
The naive algorithm for distributing the funds is to work along the outcome from the front, paying funds out to the corresponding address until no funds are remaining. In practice, we avoid introducing a dependency on the order of the payouts by using the **overlap** function `overlap`(p, ω, x), which returns the payout owed to p from outcome ω when the channel is funded with x. The overlap function can be written in python as follows:

```
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
```

general pattern is - funds deposited into a payment channel address - an outcome assigns funds from the payment channel to



- once an outcome pays out, we remove the outcome


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 }

```
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
```

We will now introduce the three on-chain operations that drive Turbo: deposit, withdrawal, and transfer.

The **deposit** operation is used to increase the funds held for a given channel. In order for the operation to be valid it must be accompanied by a transfer of funds equal to the increase.

$$D_X(x)[\![\alpha_X(y)]\!] = [\![\alpha_X(x+y)]\!]$$

The deposit can be called by anyone but a rational participant, p, should only make a deposit of size x if it causes the value of the system state for p to increase by x.

The **withdrawal** operation is used to withdraw funds held at a given address, A, by a party with a knowledge of the corresponding private key. If $x \leq x'$ then

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

In the above equation, we focus only on the affect of the withdrawal operation on the system state. In practice the withdrawal should also specify the external address that the funds should be sent to along with the signature. A potential method signature is `withdraw(fromAddr, toAddr, amount, signature)`, where `signature` is A's signature of the other parameters [2]. Note that the signature requirement coupled with the no-collision assumption means it is not possible to withdraw from the funds held for a channel.

The **transfer** operation, $T_{A,B}(x)$, is an instruction to transfer funds currently allocated to address A to address B, according to the outcome of channel A.

```
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)
```

If channel X holds x and the outcome of X allocates y to Y within x. - then update channel X to hold x, channel Y to hold y, and decrease the amount X allocates to Y by y

For example,

$$T_{A,B}(3)[\![\alpha_A(10)\beta_A(B:3, C:7)]\!] = [\![\alpha_A(7)\alpha_B(3)\beta_A(C:7)]\!]$$

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

4.1.1 Value Equivalence in Turbo

- how can we know if two system states are value equivalent? - in turbo it's simple: if there's any sequence of Ts that drive them to the same state then they're equivalent - transfer operations commute - if I can find one set of transfer operations they're equivalent

- funding a ledger channel - only deposit if it increases your value by the deposit amount - opening a subchannel

4.2 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 [3]. Following the Perun paper, we will refer to this parent channel as a **ledger channel**.

- ledger channel is assumed to be running the consensus application

4.2.1 Opening a sub-channel

- make precise the operation we performed in the introduction, where we opened a subchannel - here the enforcable state for χ is assumed to be the ready-to-fund state - doesn't matter what application is running in the channel - just what the initial balances are

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

- at each point it is value equivalent for both A and B to a b respectively - each step only changes one channel at a time - therefore we can do by single-channel-rewrite lemma

4.2.2 Closing a sub-channel

- closing a subchannel is similar - hear the finalizable state is assumed to be a conclude state from the end of the channel

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

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

where $x = a + a' + b + b'$. - the analysis is exactly the same as in the opening case

4.2.3 Topping up a ledger channel

- useful to be able to top up without disturbing sub-channels supported in a ledger channel - topping up is similar to depositing in force-move

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

- rearrange the current outcome to put the depositor last - note that $\nu_A(S_3) - \nu_A(S_2) = a'$ as it should be if A is to deposit

4.2.4 Partial checkout from a ledger channel

- partial checkout is the opposite to top up - the scenario here is two parties have a ledger channel open, supporting one or more subchannels - participant A wants to be able to withdraw - this means we need to increase the value of α_A - we will assume here that we start with $\alpha_A(0)$

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

- technique here is to create a replacement ledger channel with the updated totals and then update the original channel to point here. - in transitioning to S_4 we take L to the chain. probably using conclude to avoid the timeout

5 Nitro Protocol

- extension to turbo. introduce a different type of outcome. allows us to support channels through a third party - call these virtual channels

- example: suppose that Alice wants to play chess with Bob - doesn't have a ledger channel open but they both have a channel open with Hugo

- how this enables a hub

- how this enables multi-hop routing

5.1 Nitro Outcomes and Operations

- extends turbo by adding a new type of outcome (and a new type of channel) - refer to this as the guarantee and write this γ - guarantee allows one channel choose to pay out for certain parts of another outcome

- finalizing is exactly the same - guarantee channels can follow the same force-move transition rules and challenge etc. - interpretation of the outcomes on-chain

- in explaining how this works it will be useful to introduce another operation - cap - ensures that one allocation can't give out more to any address than another

```
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
```

- the claim operation

```
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)
```

- [diagram]: nitro state properties

5.1.1 Value Calculations

5.2 Virtual Channels

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

$$\begin{tikzcd}
& \llbracket \alpha_L(x)\alpha_{L'}(x) \rrbracket \arrow[dl, dash] \arrow[dr, dash] & \\
{[\beta_L(G:x)]_{A,C}} \arrow[d, dash] & & {[\beta_{L'}(G':x)]_{B,C}} \arrow[d, dash] \\
{[\gamma_{G,J}(\chi:x, A:x, C:x)]_{A,C}} \arrow[dr, dash] & & {[\gamma_{G',J}(\chi:x, B:x, C:x)]_{B,C}} \arrow[dl, dash] \\
& {[\beta_J(\chi:x, C:x)]_{A,B,C}} \arrow[d, dash] & \\
& {[\beta_\chi(A:a, B:b)]_{A,B}} &
\end{tikzcd}$$

Figure 1: 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

5.2.1 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

We let

$$\begin{aligned}
l_0 &= [\beta_L(A:a, C:b)]_{A,C} & l_1 &= [\beta_L(G:x)]_{A,C} \\
l_0' &= [\beta_{L'}(B:b, C:a)]_{B,C} & l_1' &= [\beta_{L'}(G':x)]_{B,C} \\
j_0 &= [\beta_J(A:a, B:b, C:x)]_{A,B,C} & j_1 &= [\beta_J(\chi:x, C:x)]_{A,B,C} \\
g &= [\gamma_{G,J}(A:x, B:x, C:x)]_{A,C} & g' &= [\gamma_{G,J'}(A:x, B:x, C:x)]_{A,C} \\
v &= [\beta_\chi(A:a, B:b)]_{A,B} & \llbracket S \rrbracket &= \llbracket \alpha_L(x)\alpha_{L'}(x) \rrbracket
\end{aligned}$$

and proceed as follows

$$\begin{array}{llllll l}
S_1 = [\![S]\!] & l_0 & l_0' & & & & \\
S_2 = [\![S]\!] & l_0 & l_0' & g & g' & j_0 & \\
S_3 = [\![S]\!] & l_1 & l_0' & g & g' & j_0 & \\
S_4 = [\![S]\!] & l_1 & l_1' & g & g' & j_0 & \\
S_5 = [\![S]\!] & l_1 & l_1' & g & g' & j_0 & v \\
S_6 = [\![S]\!] & l_1 & l_1' & g & g' & j_1 & v
\end{array}$$

- 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

5.2.2 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