

Virtual state channels are simple and useful

Andrew Stewart, Mike Kerzhner, George Knee, Sebastian Stammer, Matthias Who, *

November 11, 2021

Abstract

Virtual state channels allow peers to bootstrap existing connections to form a state channel network. Existing virtual channel constructions are not so good. We present an amalgamation of two existing state channel protocols, Nitro and Perun, which considerably improve the practical application of virtual state channels..

1 Introduction

1.1 State channels

Introduce state channels.

1.2 Prior work

Review existing protocols

- Perun: Too many channels, recursive construction, intermediaries block ledger updates.
- Nitro: Even more channels, still recursive construction.
- Donner: UTXO model, complicated construction, authors have not yet understood it. We suspect there is a mapping between ATP and Donner.
- See Perun paper for more background.

1.3 Our contribution

In this paper we present Asset Transfer Protocol¹, a simple and practical protocol for constructing state channel networks. We give a detailed description of

*The expanded list of authors represent the collaborative nature by which this protocol was developed.

¹ATP, or adenosine tri-phosphate, serves as a reasonable analogy for Asset Transfer Protocol – both enable a burst of high-performance activity, followed by periodic replenishing of reserves. Perun, the Norse God of lightning, is partially responsible for getting nitrogen into mitochondria.

how to construct ledger channels and virtual channels, including how to safely open and close these channels entirely off-chain. Taken with our earlier work on Perun and Nitro, this paper gives a complete specification for building a state channel network capable of running arbitrary state channel applications.

Citations
needed

- We describe a simple protocol for quickly constructing virtually funded channels in an adversarial state channel network.
- The idea is: Perun laid out the correct data structures, and Nitro introduced guarantees. By combining the two ideas, both protocols are significantly simplified.
- Quick means there are few serial messages required in the protocol.
- We prove it's safe for all parties involved.
- In the appendix, we outline a streamlined ledger funding protocol.
- We discuss auxilliary protocols (top-ups, partial checkouts, etc) in an extended version of the paper.

Nitro uses
"ledger"
differently
than Perun.
What's a
good term
to use here?

An implementation of this protocol is underway in Golang and Solidity, under a grant from the Filecoin Foundation and support from Consensys Mesh. ATP channels will alleviate key UX issues in the Filecoin Retrieval Market. The ability to use bespoke application logic will enable the retrieval market to use novel cryptoeconomic incentives to reward retrieval miners.

add ref

1.4 Outline

- ?? - specification of on-chain components
- ?? review of ledger funded channels
- ?? review of ledger funded channels
- ?? statement of theorem + proof

fix references

2 On-chain protocol

2.1 Channel attributes

A channel X has the following constant attributes:

- peers, an ordered list of signing keys
- appDef, an address defining the location of the channel's application logic.
- nonce, a nonnegative integer
- challengeDuration, a nonnegative integer

Title

Describe on-chain model
(smart contracts, etc.)

The channel's id is computed as `hash(peers, appDef, nonce)`. The inclusion of a nonce allows for a fixed set of peers to construct an arbitrary number of distinct channels.

Should we define these data structures as a protobuf or something?

Channels also have the following variable attributes:

- version, a nonnegative integer
- outcome, specified in Subsection 2.4
- appData, unspecified bytes parsed by custom application logic
- isFinal, a boolean flag

Together, constant and variable attributes form a **state**.

A channel has an on-chain **adjudication state**, with the following attributes:

- holdings, a nonnegative integer representing the cumulative deposits into the channel
- version, a nonnegative integer
- outcome, specified in Subsection 2.4
- finalizationTime, a timestamp indicating the time at which the channel is considered finalized.

The adjudicator stores a mapping from channel id to adjudication state.

2.2 Applications

A state channel application is a smart contract implementing a `latestSupportedStatefunction` with the signature `. The variable part returned is assumed by the adjudicator to be the most recent version of the channel's state to be supported by all peers in the channel.` ²

FILL

The most basic application, coined a **consensus app**, follows the following specification:

- Revert if `states.length` \neq 1.
- Revert if `states[0]` is not signed by all of `fixedPart.peers`.
- Return `states[0]`.

In other words, a consensus application is used to ensure that all peers support the unique state provided to the adjudicator.

²Note that an application is free to define support in an arbitrary manner. For instance, an application where assets flow unidirectionally from Alice to Bob may specify that Alice can unilaterally support non-final states, and Bob can unilaterally transition from a non-final state signed by Alice to a final state signed by Bob. Care must be taken to ensure application rules encode the fair distribution of assets.

A more sophisticated application is specified in ??.

fix ref

2.3 Adjudication

A state channel protocol assumes an adversarial setting. To protect against arbitrary behaviour among peers, an adjudicator implements a **challenge** operation, enabling peers to recover funds from the channel after a timeout.³ It is implemented according to the following specification:

- Check that the channel is not finalized, ie. $\text{statusOf}(X).\text{finalizationTime} \geq \text{now}$.
- Let $s = \text{appDef.latestSupportedState}(a, b)$
- Set $\text{statusOf}(X)$ to $(\text{foo}, \text{bar}, \text{baz})$

fix

As timers significantly worsen user experience, we also describe a collaborative operation **conclude**, which instantly finalizes a channel.⁴

- Revert if X is finalized, ie. $\text{statusOf}(X).\text{finalizationTime} \geq \text{now}$.
- Let $s = \text{appDef.latestSupportedState}(a, b)$.
- Revert if $s.\text{isFinal}$ is false.
- Set $\text{statusOf}(X)$ to $(\text{foo}, \text{bar}, \text{baz})$

Make sure this is necessary in the paper?

fix ref

fix

2.4 Outcomes and Asset Management

Define deposits

An **allocation** $\text{Alloc}(A, a)$ is a data structure encoding a destination A and an amount a . A **guarantee** $\text{Guar}(X, x, [[A_1, A_2, \dots, A_k]])$ encodes a target X , an amount x , and an ordered list of destinations A_1, \dots, A_k . An **exit** is either an allocation or a guarantee. An **outcome** is an ordered list of exits.

Outcomes are in priority order, so that if the channel does not hold enough funds to pay all the coins that are due in its finalized outcome, then the exits at the beginning of the allocation will be triggered.⁵ 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.

missing refs

is this used?

Allocation exits are triggered via the **transfer** operation, $\text{Transfer}(A, i)$, which follows the following specification:

1. Reverts if the channel A is not finalized.

This figure needs to be updated

³In practice, a state’s support may need to be provided over multiple blockchain transactions to account for bounds on computation complexity. We ignore this detail.

⁴Implementations may also specify a **checkpoint** operation, which reverts for finalized channels or when presented with a stale state, and otherwise replaces the latest outcome and cancels any existing timer. See ?? for details

⁵This choice is somewhat arbitrary yet practical. Alternatively, unordered outcomes which use additional state to manage deposits can enable concurrent deposits with less off-chain overhead, at the expense of increased cost to users. See Vector ?? or Perun ??.

address

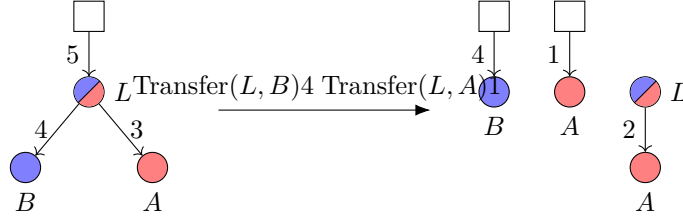


Figure 1: A graphical representation of the outcome $[Alloc(B, 4), Alloc(A, 3)]$. All exits are allocations, meaning Outcomes pay out in priority order. In the diagram, B is drawn to the left of A to show that B has higher priority in the outcome of L. In this example, L can afford 4 coins for B, but can only afford 1 coin for A.

2. Reverts if the i -th exit e in A's outcome is not an allocation.
3. Sets x to be $e.amount$.
4. Reduces the funds held in channel A by x .
5. Sends x coins to B.
6. Reduces $e.amount$ by x .

Guarantees are triggered via the **claim** operation $Claim(A, i)$, which follows the following specification:

1. Reverts if the channel A is not finalized.
2. Reverts if the i -th exit e in A's outcome is not a guarantee
3. Reverts if the channel $e.target$ is not finalized.
4. Reduces the funds held in channel A by x .
5. Sends x coins to the ether
6. Reduces $e.amount$ by x .

address
e.amount
issue

decide where
to fetch x
from

address

fill this in

address
e.amount
and e.target
issue

Add a table
outlining on-
chain state
transitions

2.5 Recap

3 Off-chain protocols

3.1 Ledger Channels

A **ledger** channel is a channel which is funded directly by the ledger.

3.1.1 Depositing into a ledger channel

(Provide minimal explanation, and refer to the Nitro whitepaper for details.)

3.1.2 Withdrawing from a ledger channel

3.2 Virtual channels

Suppose we have peers $A = P_0, P_1, \dots, P_n, P_{n+1} = B$ where:

- each (P_i, P_{i+1}) pair already has a ledger channel L_i running the consensus app
- Alice (P_0) and Bob (P_{n+1}) want to make (virtual) payments between each other.

We can safely fund a joint channel J with the following protocol:

Round 1: Each participant signs a state s for J with $turnNum = 0$ and outcome $\{A : a_0, B : b_0\}$. They sign s and send

Round 2: For each $i = 0, \dots, n$, participants P_i and P_{i+1} sign an update in L_i to:

- deduct a_0 from P_i 's balance in L_i
- deduct b_0 from P_{i+1} 's balance in L_i
- include the allocation $G_i = J : amount : x, left : P_i, right : P_{i+1}$ where $x = a_0 + b_0$

For instance, L_i 's outcome might change


- from $[\text{Alloc}(P_i, bal_i), \text{Alloc}(P_{i+1}, bal'_i), \text{Guar}(X', x', [foo, bar])]$
- to $[\text{Alloc}(P_i, bal_i - a_0), \text{Alloc}(P_{i+1}, bal'_i - b_0), \text{Guar}(X', x', [P_i, P_{i+1}]), \text{Guar}(X, x, [P_i, P_{i+1}])]$.


Round 3: Alice blocks until she receives a counter-signed update in L_0 . Bob blocks until he receives a counter-signed update in L_n . For $i \in \{1, \dots, n\}$, P_i blocks until they have counter-signed updates in L_i .

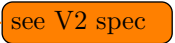
Each participant signs a post-fund state s_1 for J with $version = 1$ and outcome $\{A : a_0, B : b_0\}$.

For each peer, the protocol is completed once a full set of signatures is received on s_1 . At this point,

- each P_i has $a_0 + b_0$ fewer tokens across their two ledger channels L_{i-1} and L_i
- Alice (P_0) has a_0 fewer tokens in L_0
- Bob (P_{n+1}) has b_0 fewer tokens in L_n
- every participant's ledger channel reductions are offset by an equal allocation to the joint channel J
- J 's outcome

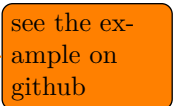
To secure this protocol, we now specify application rules for J . 

We are now ready to state the main result of this paper: 

Proof: 

Case 1: Case 2: Case 3:

3.3 Variations

3.3.1 Generic virtual channels. 

3.4 Reduced latency of construction. 