Virtual state channels are definitely simple and probably useful

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

November 15, 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 considerately 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 Protocol[1], 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.

[1]ATP, or adenosine tri-phosphate, serves as a reasonable analogy for Asset Transfer Protocol – both enable a burst of high-performance activity, followed by periodic replentishing of

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.

  Nitro uses “ledger” differently than Perun. What's a good term to use here?
- We discuss auxilliary protocols (top-ups, partial checkouts, etc) in an extended version of the paper.

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

Mention Groethendic's quote about making things trivial using the correct definitions.

2 On-chain protocol

fix references

Title

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

2.1 Channel attributes

A channel 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

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?

A **channel state** is comprised of channel constants together with 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

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 two values for a given channel X:

- holdings(X), representing the sum total of the deposits into the channel.
- statusOf(X), representing an 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.

reserves. Perun, the Norse God of lightning, is partially responsible for getting nitrogen into mitochondria.

[2]Note 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.

- Return states[0].

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

A more sophisticated application is specified in Section 3.2.

2.3 Adjudication

A state channel protocol assumes an adversarial setting. Therefore, assets are deposited into an adjudicator contract, which releases funds.

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

$$(s.\text{outcome}, s.\text{version}, \text{finalizationTime} = \text{now} + s.\text{challengeDuration})$$

fix

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

Make sure this is necessary in the paper?

- Revert if X is finalized, ie. $\text{statusOf}(X).\text{finalizationTime} \geq \text{now}$.
- Let $s = \text{appDef.latestSupportedState}(a, b)$.

fix ref

- Revert if $s.\text{isFinal}$ is false.

fix

- Set $\text{statusOf}(X)$ to $(s.\text{outcome}, s.\text{version}, \text{finalizationTime} = \text{now})$

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

Outcomes are in priority order

This is irrelevant to the paper.

[3] In 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.

[4] Implementations 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

In this paper, we assume that ledger channels are fully funded when an exit is triggered, ie.

$$\text{holdings}(X) = \sum_{e \in \text{statusOf}(X).\text{outcome}} e.\text{amount}.$$

There are multiple ways of implementing deposits, and the main result **??** does not depend on this choice.

missing ref

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

1. Reverts if the channel A is not finalized.
2. Sets $e = \text{statusOf}(X).\mathit{outcome}[i]$.
3. Reverts if e is not an allocation.
4. Sets x to be $e.\mathit{amount}$.
5. Reduces the funds held in channel A by x.
6. Sends x coints to B.
7. Sets $e.\mathit{amount} = 0$.

address

address e.amount issue

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 $e.\mathit{target}$ is not finalized.
4. Reduces the funds held in channel A by x.
5. Sends x coints to the ether
6. Reduces $e.\mathit{amount}$ by x.

If we call this *reclaim instead of claim, and simply modify outcomes, we'll end up with a cleaner statement.*

decide where to fetch x from

address

fill this in

address e.amount and e.target issue

2.5 Recap

3 Off-chain protocols

Add a table outlining on-chain state transitions

3.1 Ledger Channels

A **ledger** channel is a channel which is funded directly by the ledger. For simplicity of discussion, we assume that ledger channels operate under the consensus app described in Subsection 2.2, and assume that peers.

This choice is inefficient, and provides suboptimal time to paymentin the worst case, since. Efficient designs which achieve best-possible results even in the worst case appear to be viable. Their practical implementation is a problem of current research.

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 $A = P_0, P_1, ..., P_n, P_{n+1} = B$ 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 $[\text{Alloc}(A, a_0), \text{Alloc}(B, b_0)]$. They sign s and send to each participant.

Round 2: For each i = 0,...,n, participants P_i and P_{i+1} sign and exchange 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 guarantee $G_i = \text{Guar}(J, x, [P_i, P_{i+1}])$

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 $i \in \{1, \ldots, n\}$, P_i blocks until they have counter-signed updates in L_i.

Once unblocked, each participant signs a post-fund state s_1 for J with version = 1 and outcome $[\text{Alloc}(A, a_0), \text{Alloc}(B, b_0)]$.

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

Before securing this protocol, we observe some properties of this protocol and its derivatives:

- The happy path requires $O(n)$ network overhead and $O(1)$ time to complete across n intermediaries. This improves on **??**, and matches **??**.
- In the event of an unresponsive or malicious peer, exactly one participant has to launch a challenge for the J channel. Only peers "connected" to the faulty peer need to challenge in their ledger channel with their peer. Thus, we acheive the same sad-case complexity as **??**.
- In a unidirectional virtual channel – one where Bob initially deposits 0 – it is possible to eliminate Round 3.
- At least in the case of one intermediary, rounds 1 & 2 can be partially combined – see section 3.4. The end result is, Bob can redeem payments from Alice after **two sequential networking messages**. As far as we are aware, this is state of the art, and appears to achieve a theoretical minimum.

reference perun

reference Donner

missing ref

Explain how

fix ref

Conjecture 3.1. A trustless virtual state channel protocol requires at least two sequential network messages to be funded.

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

specify

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

Theorem 3.1. For each i, if P_i

Proof:

see V2 spec

Case 1: Case 2: Case 3:

3.3 Variations

3.3.1 Generic virtual channels.

see the example on github

3.4 Reduced latency of construction.