SATP: A simple and scalable protocol for virtual state channel networks

Andrew Stewart*[1], Colin Kennedy[1], Mike Kerzhner[1], George Knee[1], Matthias Geihs[2], and Sebastian Stammler[2]

[1]Magmo (Consensys Mesh)
[2]PolyCrypt GmbH

September 28, 2022

Abstract

Virtual state channels allow peers to bootstrap existing connections to form a state channel network. We present Stateful Asset Transfer Protocol (SATP[1]), an amalgamation of two existing state channel protocols, Nitro and Perun, which considerately improves the practical application of virtual state channels, evangelizing an approach of security-by-simplicity. In special cases, we conjecture to have achieved theoretical optimal performance.

1 Introduction

1.1 State channels and scaling

Among scaling techniques for distributed ledgers State Channels offer best-in-class transaction throughput but suffer from some worst-in-class usability issues. In the naive State Channel construction, a set of participants initiate a channel via an on-chain transaction, transact indefinitely with one another off-chain at speed and cost of normal network operations, and conclude their interaction with a second on-chain transaction. **The good**: the costs of the two on-chain transactions are amortized over the practically unlimited number of off-chain transactions. **The bad**: off-chain interactions are limited to the specific persons who initiated the channel, and limited to the exchange of the specific capital lockup defined in the channel's opening transaction. **The ugly**: networking costs scale up in the square of the number of participants in a state channel,

*andrew.stewart@mesh.xyz

[1]Simulation of Adenosine Tri-Phosphate is a tempting backronym. ATP serves as a reasonable analogy for state channels – 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.

and channels become less responsive as the number of potential failure points increases.

Still, use cases exist where the extremely high throughput to cost ratio of a State Channel is a practical requirement: streaming payments in exchange for access to a video or audio stream, SaaS access to APIs that might get hit tens of thousands of times per day, et cetera. This need has motivated the development of State Channel Networks, which improve on the naive construction by allowing participants to route transactions to one another through a path of intermediaries.

1.2 State channel preliminaries

A state channel (henceforth channel) is a container of **versioned states** which are cryptographically committed to. States consist of a) an **outcome**; b) some **data**; and c) some **rules** which govern how the outcome and data may be updated (i.e. committed to with a greater version) by a fixed number of **peers** (often two).

Commitments take the form of digital signatures on states.

Taken together, the rules and data constitute an **application** that is said to run inside the channel. For example, the rules might dictate that the outcome may only change in such a way so as to increase the assets that eventually flow to a certain peer. In such a case, we recover a **payment channel** as a special case of a state channel. An example of the more general extension would be to encode (for example) the rules of Chess, and to have the outcome update only to a distribution of assets fairly reflecting (for example) the current value of the pieces on the board.

The outcome dictates how assets, initially locked into some secure ledger when the channel is **funded**, will be redistributed when it is **de-funded**. It specifies a number of **balances**, each of which allocates assets to various **destinations** – which may include the channel peers, other addresses in the ledger, or other channels. The ledger could be a blockchain such as Bitcoin or Ethereum, or indeed another state channel (considering a channel's outcome to constitute a private ledger of its own).

A **state channel network** typically has one or more **adjudicator** smart contracts deployed to one or more distributed ledgers or blockchains. The adjudicators hold assets for a graph of interconnected channels. In this work, we concentrate on **ledger channels**, which are typically funded by the underlying ledger, and **virtual channels**, which are funded by ledger channels (and therefore exist at a higher level of abstraction). As we shall explain in further detail, virtual channels are also typified by being funded by more than one ledger channel, each with an overlapping but distinct set of peers.

In this work, the state channel rules consist of a function accepting an arbitrary number of states and signatures and returning the latest single state which is deemed to be **supported** by those inputs. The inputs constitute what we call a **support proof**. A supported state will be accepted by the adjudication contract – with greater versions taking precedence over lesser ones – and its outcome can then be executed.

The chief characteristic of a state channel is that state and value may be updated and transferred without affecting the underlying ledger. They achieve this by dint of the fact that the updates are able to eventually unlock assets on the underlying ledger at some point in the future, under mild assumptions. Peers can readily verify this fact. Such state updates are therefore often termed "off-chain", and avoid some of the disadvantages of directly updating the underlying ledger. For example, updating the underlying ledger typically requires the payment of fees proportional to the computational and storage load implied by those updates – a load known as **gas** in the Ethereum ecosystem. State channels continue to be liable for costs associated with residual gas – that is, the costs of locking and unlocking assets as well as adjudicating disputes.

For the purposes of this paper, the lifecycle of a state channel is as follows: a channel is **opened**, funded, **closed** and de-funded. At any point, the channel may enter a **dispute**.

A dispute happens when a peer wishes to force the channel to close on chain so it may be de-funded. The peer will submit a support proof to the adjudicator contract, which will set a timeout. The timeout allows other peers to counter the dispute with alternative support proofs. Eventually, the channel is closed and may be de-funded according to the **latest** supported state (i.e. the supported state with the greatest version.

A State Channel Network may be theoretically secure, yet the cost to recover funds via disputes is a large percentage of or may even exceed the total recoverable funds. Therefore, we introduce the informal concept of the **economic security** of a protocol, which increases as the amortized gas cost to recover funds across channels decreases.[2]

1.3 Prior work

Early work on payment channel networks was done in 2016 with Lightning [2], whose goal was to scale Bitcoin by enabling multi-hop, off-chain payments. It involves intermediaries on the critical path of a stream of payments between two users, placing intense demand on the liquidity provider when providing routing services for multiple streams.

Virtual channels enable two users, Alice and Bob, to transact directly, privately, and off-chain, by leveraging existing connections to a common intermediary Irene. Counterfactual [3], Nitro [4] and Perun [5, 6] are existing state channel protocols which enable virtual channels. These protocols were featured

[2]We promote the practice of tracking gas costs of recovery mechanisms in source code [1].

in Web3Torrent [7], a POC application designed to incentivize torrent seeders through micropayments.

In Ref. [8] Perun improved on the worst-case time and gas-cost complexity required to resolve disputes across multiple intermediaries,. Donner [9] achieves similar results in a UTXO-based model, and also improves on the time-complexity required to open a virtual channel across n hops.

1.4 Our contribution

We merge[3] the virtual channel constructions of Perun and Nitro, developing a protocol which more efficiently constructs a virtual channel V involving peers $P_0 = \texttt{Alice}, P_1, \ldots, P_n, P_{n+1} = \texttt{Bob}$. V is funded via pre-existing channels L_i between P_i and P_{i+1}, which we assume to be fully-funded as a precondition throughout this paper.

We show that V can be securely funded via $O(1)$ rounds by using Protocol 3.1. In special cases, we reduce this number to 2 rounds, which we conjecture to be the minimum possible.

Observe that Donner [9] achieves similar results in a UTXO-based virtual channel construction.

Open Question 1.1. *Are Protocols 3.1-3.2*[4] *substantially different from Donner?*

Once opened, Alice and Bob can transact privately in V, requiring no further participation from the intermediaries $P_1, \ldots, P_n$. When Alice and Bob desire to "settle the balances", funds are redistributed across the ledgers $L_0, \ldots, L_n$ in Protocol 3.2.

In case of misbehaviour, funds that have been allocated to V can be recovered via Protocol 3.3. Its worst-case time complexity is $O(1)$, and its worst-case gas-cost is $O(k)$, where k is the number of misbehaving peers.

Theorem 3.2 ensures that intermediaries' funds are never at risk, under the following two **security assumptions**:

- (Liveness) Any peer can successfully record transactions on the blockchain in $O(1)$ time.
- (Safety) Forging cryptographic signatures is infeasible.

Our focus is on carefully using appropriate primitives to specify, in plain English and "Solidity/Typescript-inspired pseudocode", our virtual channel protocol in

[3]The Solidity implementations of Perun and Nitro bear a stronger resemblance than their theoretical specifications.

[4]These protocols were discovered independently from Ref. [9], and may therefore be substantially different.

a way that facilitates easy security analysis.[5][6] This naturally accomplishes our main goal, which is to implement an optimized, secure virtual channel protocol in Solidity.

Magmo and Polycrypt are jointly implementing this protocol in Go, targeting the Filecoin Retrieval Market [12]. We anticipate a possible network topology involving a *dense "core" of liquidity providers facilitating channels between a large set of providers and a very large set of clients.* In this topology, a low-overhead construction of 2-intermediary virtual channels reduces the latency experienced by users. We further anticipate using bespoke "application channels" to enable the retrieval market to use novel cryptoeconomic incentives to reward retrieval miners.

2 On-chain protocol

2.1 Channel attributes

A channel has two key types of objects, **constants** and **variables**.

```
interface ChannelConsts {
  peers: PublicKey[]
  appDef: Address
  nonce: uint
  challengeDuration: uint
}

interface ChannelVars {
  version: uint
  isFinal: boolean
  outcome: Outcome // specified in Section 2.4
  appData: bytes   // unspecified, parsed by custom app logic
  signatures: Signature[]
}
```

The channel's id is computed from its constants 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.

A **channel state** comprises its constants together with a set of variable attributes:

```
interface ChannelState {
  consts: ChannelConsts
  vars: ChannelVars
}
```

When a channel dispute is being handled by the adjudicator, the following attributes are recorded:

[5]Our eyes are set on a formal TLA+/equivalent specification. See [10] for successful anecdotes from industry and [11] for a successful anecdote from Magmo.

[6]We recall Grothendieck saying something along the lines of "with the correct definitions, the proof is obvious", but cannot find a reference.

```
interface AdjudicationState {
  version: uint
  outcome: Outcome
  finalizationTime: uint
}
```

The adjudicator stores adjudication states in a mapping `statusOf`, using channel ids as keys.

2.2 Applications

A support proof is an array of states, each state containing one or more signatures from the channel's peer list. The following data structure efficiently packages this data:

```
interface SupportProof {
  consts: ChannelConsts,
  vars: ChannelVars[]
}
```

A state channel application is a smart contract implementing the following signature:

```
function latestSupportedState(proof: SupportProof): ChannelState
```

We rely on two special applications for the main results of this paper.

The variable part returned is understood by the adjudicator to be the most recent version of the channel's state to be supported by all peers in the channel, even if not explicitly signed by every peer. [7]

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

```
function latestSupportedState({const, vars}) {
    require(vars.length == 1)
    require(signedByEveryone(vars[0], consts.peers))
    return {consts, vars: vars[0]}
}
```

In other words, the consensus application is used to ensure that all peers have seen and agree with the unique state provided to the adjudicator. A more sophisticated application, used to construct virtual channels, is specified in Section 3.2.

[7]Note that an application is free to define support in an arbitrary manner. For instance, an application where assets flow in one direction from Alice to Bob may specify that Alice can unilaterally support a non-final state with version 0, and Bob can unilaterally transition from any non-final state signed by Alice to a final state, with the same outcome, signed by Bob. Care must be taken to ensure application rules encode the fair distribution of assets.

2.3 Adjudication

A state channel protocol assumes an adversarial setting. Therefore, assets deposited into an adjudicator contract are released only when given explicit consent on the final outcome by the channel's participants. As timeouts significantly worsen user experience, adjudicators typically allow peers to collaboratively conclude a channel without delay, by supporting a state with `isFinal` = `true` and submitting the support proof to a `Conclude` operation.

To protect against arbitrary behaviour among peers, an adjudicator implements a **challenge** operation, enabling peers to recover funds from the channel after a timeout.[8] It is implemented according to the following specification:

```

function isOpen(status: AdjudicatorStatus): bool {
    // the channel has no active challenge or the channel has not
    // yet finalized
    return status.finalizesAt == 0 || status.finalizesAt > now
}

function challenge({consts, vars}: SupportProof) {
  let id = channelId(consts)
  let adjudicatorStatus = statusOf(id)
  require(isOpen(adjudicatorStatus))

  let supportedStateVars = Application.at(state.appDef).
      latestSupportedState(consts, vars).vars

  require(supportedStateVars.version >= adjudicatorStatus.version)

  statusOf(id) = {
    outcome: supportedStateVars.outcome,
    version: supportedStateVars.version,
    finalizesAt: now + consts.challengeDuration,
  }
}
```

Listing 1: *"Challenge"*

2.4 Outcomes and Asset Management

An **allocation** is a data structure $\mathcal{A}(a)$ representing an amount a to be paid to a destination.

A **guarantee** is a data structure $\mathcal{G}(x, [A, B])$ representing an amount x and an ordered list of destinations A and B.

We slightly abuse notation, and say that `item.amount` $= x$ when either `item` $= \mathcal{A}(x)$ or `item` $= \mathcal{G}(x, [A, B])$.

[8]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.

An **outcome** is an *ordered* dictionary, mapping a destination to either an allocation or a guarantee. [9] As an example, a channel finalized with outcome

$$\{\texttt{Alice} : \mathcal{A}(a), \texttt{Bob} : \mathcal{A}(b), V : \mathcal{G}(x, [A, I])\}$$

would allow a tokens to be withdrawn by Alice, b tokens to be withdrawn by Bob, and x tokens to be *reclaimed*, using the "finalized" outcome of V. Keys are ordered from left to right as they appear in the text: in the outcome $o = \{A : \mathcal{A}(a), X : \mathcal{G}(x, [A, I]), B : \mathcal{A}(b)\}$, for instance, $o[0]$ is $o[A]$, $o[1]$ is $o[X]$, and $o[2]$ is $o[B]$.

Guarantees are reclaimed via the **reclaim** operation `Reclaim`(L, V)[10], which follows the following specification:

```
function reclaim(ledgerChannelId, targetChannelId) {
  let ledgerStatus = statusOf(ledgerChannelId)
  let targetStatus = statusOf(targetChannelid)

  require(0 < ledgerStatus.finalizesAt <= now)
  require(0 < targetStatus.finalizesAt <= now)

  let guarantee = ledgerStatus.outcome[targetChannelid]
  require(isGuarantee(guarantee))
  let [left, right] = guarantee.peers

  let o_ledger = ledgerStatus.outcome
  let o_target = targetStatus.outcome

  o_ledger[left] += o_target[0]
  o_ledger[right] += o_target[1]

  delete o_ledger[targetChannelId]

  statusOf(ledgerChannelId).outcome = o_ledger
}
```

Listing 2: *"Reclaim"*

The `Reclaim` operation enables intermediaries in a virtual channel construction to "partially fund" the virtual channel without risking funds, freeing them to sign ledger channel updates in any order – this is key to securing Protocol 3.1.[11]

3 Off-chain protocols

3.1 Ledger Channels

A **ledger** channel is an "auxiliary channel" used as a private ledger in various protocols. For instance, a ledger channel L may include an allocation $C : \mathcal{A}(c)$

[9] In practice, the order may also dictate which destinations receive priority in case the channel is insufficiently funded.

[10] A gas-optimized `Reclaim(X)` might reclaim all reclaimable guarantees in X's outcome.

[11] This decoupling of ledger channel updates is the main contribution of Nitro protocol.

for some channel C. In this paper, we assume as a precondition that all ledger channels are fully-funded – each item listed in its outcome.

For simplicity of discussion, we assume that ledger channels operate under the consensus app described in Subsection 2.2. [12]

3.2 Virtual channels

Definition 3.1. A peer **commits** to a state s in a channel by signing s and sending it and the resulting signature to each peer in $s.\texttt{consts}.\texttt{peers}$. They then block until they have collected enough signatures $\texttt{sigs}$ on s such that $\{\texttt{consts} : s.\texttt{consts}, \texttt{vars} : \{...s.\texttt{vars}, \texttt{sigs}\}\}$ is a support proof. At this point, the state s has been **committed**.

A peer who has signed state s with version n and refuses to sign other states until they hold a support proof for s is said to be **blocking.**

Suppose we have peers $\texttt{Alice} = P_0, P_1, ..., P_n, P_{n+1} = \texttt{Bob}$ where:

- for $i = 0, \dots, n$, there exists a ledger channel L_i between P_i and P_{i+1}, running the consensus app
- Alice (P_0) and Bob (P_{n+1}) want to make (private) payments between each other.

We can securely fund a virtual channel V with the following protocol:

Protocol 3.1 (Opening a virtually funded channel)**. Round 1**: Each participant commits to a "pre-fund" state s_0 for V with $\texttt{version} = 0$ and outcome $\{A : \mathcal{A}(a_0), B : \mathcal{A}(b_0)\}$.

Proceed when s_0 is committed.

Round 2: For each i = 0,...,n, participants P_i and P_{i+1} commit an update in L_i to deduct a_0 from P_i's balance, b_0 from P_{i+1}'s balance, and include the guarantee $G_i = V : \mathcal{G}(a_0 + b_0, [P_i, P_{i+1}])$ – in L_i, a_0 has been debited to V by P_i and b_0 debited by P_{i+1}.

For example, if L_i's outcome is

$$\{P_i : \mathcal{A}(bal_i), P_{i+1} : \mathcal{A}(bal'_i), V' : \mathcal{G}(x, [P_i, P_{i+1}])\}$$

it would change to

$$\{P_i : \mathcal{A}(bal_i - a_0), P_{i+1} : \mathcal{A}(bal'_i - b_0), V : \mathcal{G}(a_0 + b_0, [P_i, P_{i+1}]), V' : \mathcal{G}(x, [P_i, P_{i+1}])\}.$$

Round 3: Each participant commits a "post-fund" state s_1 for V which is identical to s_0, besides setting $\texttt{version} = 1$.

[12]This choice is inefficient in the worst case, since new updates cannot be proposed until in-flight updates are resolved. Efficient designs which achieve best-possible results even in the worst case appear to be viable, and their practical implementation is a problem of current research.

Explanation: Round 1 enables any peer to record a predictable outcome for V on-chain. The application rules for V must prevent it from finalizing with any outcome until Round 3 has terminated. Round 2 allocates funds to V, relying on the consistency of V's unique "finalizable" outcome to protect intermediary's funds.

At this point,

- each P_i has $x = a_0 + b_0$ 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
- each ledger channel has had x total tokens deduced from its allocations, and includes a guarantee targeting the virtual channel V for amount x.

Protocol 3.1 requires $O(n)$ network overhead and $O(1)$ time to complete across n intermediaries. This improves on [8], and matches [9].

In a unidirectional virtual channel – one where Bob initially deposits 0 – it is possible to entirely eliminate Round 3 from Protocol 3.1. At least in the case of one intermediary, rounds 1 & 2 can be partially combined.[13] The end result is, Bob can redeem an initial payment from Alice after **two rounds**. As far as we are aware, this is state of the art. We conjecture that this achieves a theoretical lower bound.

Conjecture 3.1. *A state channel protocol requires at least two rounds to securely fund a virtual channel.*[14]

Once Alice and Bob are finished the channel, they may collaboratively conclude V, and remove the guarantees according to the following protocol.

Protocol 3.2 (Collaboratively closing a virtually funded channel)**. Precondition**: Alice and Bob collaboratively agree to conclude V with outcome $o = \{A : \mathcal{A}(a), B : \mathcal{A}(b)\}$.

Round 1: All peers in V commit to s_f with `version` $= 4$ and outcome o.

Round 2: For $i = 0, \ldots, n$, all peers in L_i commit to a state in L_i which increments `version`, removes the guarantee G, adds a to G.`peers`[0]'s balance and adds b to G.`peers`[1]'s balance – in L_i, a has been credited to P_i by V and b credited to P_{i+1}.

[13]This is (currently) left as an exercise for the reader.

[14]Informally, this conjecture can be restated as follows: Suppose Alice and Irene have a ledger channel, and Irene and Bob have a ledger channel. We do not believe there is any way for Alice to securely convince Bob that Irene will foot the bill at the closure of a virtual channel, without Bob seeing at least one signature from Irene. Since Alice must first ask Irene for a signature, this necessitates at least two rounds.

Protocols 3.1-3.2 are peer-to-peer protocols, and may break down for a multitude of reasons. If a peer detects misbehaviour[15], they may take unilateral action to reclaim funds locked up with a virtually funded channel:

Protocol 3.3 (Unilaterally reclaiming funds from V)**.**

1. Challenge in V, if possible, with the latest known supported state.
2. Challenge in each of L_{i-1} and L_i, precisely when there's a possibility of L being finalized with a guarantee $V : \mathcal{G}(x, [\ldots])$ in its outcome.
3. After timeouts (1) and (2), reclaim any guarantees in L_{i-1} or L_i pertaining to V.

We omit the proof of the following safety guarantee, which takes advantage of the fact that exactly one participant has to launch a challenge for V, and only peers "connected" to the misbehaving peer P need to challenge in their ledger channel with P.

Theorem 3.1. *Protocol 3.2 consumes $O(1)$ gas and terminates in time*

$$O(\max(L_{i-1}.\texttt{challengeDuration}, L_i.\texttt{challengeDuration}) + V.\texttt{challengeDuration}).$$

Thus, Protocol 3.3 achieves the same sad-case time-complexity as [8]. We anticipate a careful implementation of Protocol 3.3 will demand significantly less gas, leading to higher economic security.

The main security guarantee we provide for this protocol states that intermediaries' balances are invariant and are not locked up indefinitely in the virtual channel. We omit the analogous statement that can be stated for the end-users Alice and Bob.

Theorem 3.2. *Let $d(V, P_i)$ be the amount debited to V from P_i's balances, across L_i and L_{i-1} in Protocol 3.1.*

Let $c(V, P_i)$ be the amount credited to P_i's balance from guarantees targeting V across L_i and L_{i-1} in both Protocol 3.2 and 3.3.

For all V, for all $i = 1, \ldots, n$, if P_i follows Protocols 3.1-3.3, then $c(P_i) = d(P_i)$.

The proof relies on specific virtual channel rules described in Appendix A. We record the following key fact from Lemma A.1:

Lemma 3.1. *Suppose V finalizes on-chain. Either V is finalized with $\texttt{version} = 0$ and its original outcome $\{A : \mathcal{A}(a_0), B : \mathcal{A}(b_0)\}$ or it is finalized with an outcome $\{A : \mathcal{A}(a), B : \mathcal{A}(b)\}$, where $a_0 + b_0 = a + b$.*

Proof of Theorem 3.2.

[15] We allow for a flexible interpretation of *misbehaviour* – for instance, intermediaries may abandon Protocol 3.1 after some timeout.

Case 1 (Round 1 of Protocol 3.1 has not finished). In this case, P_i cannot distinguish between "V has not yet finalized" and "V will never finalize". This does not matter, since until Round 1 terminates, P_i will never begin committing updates in either L_i or L_{i-1} that include a guarantee targeting V.

Without P_i's signature on such an update, the consensus app guarantees there is no risk to P_i's balances in L_i or L_{i-1}.

Case 2 (Round 1 of Protocol 3.1 has finished, but Round 2 has not.). L_i and L_{i-1} may be both finalized on-chain, but it is out of P_i's control whether they are finalized with guarantees targeting V. However, V has necessarily finalized with $\texttt{version} = 0$ and therefore with outcome $\{A : \mathcal{A}(a_0), B : \mathcal{A}(b_0)\}$ by Lemma 3.1.

If L_i's outcome does not include G_i, there are no funds to reclaim. Otherwise,

- P_i's balance in L_i has decreased by a_0 in Round 2 of Protocol 3.1.
- $\texttt{Reclaim}(L_i,)$ will increase P_i's balance in L_i by a_0.

Similar analysis holds for L_{i-1}.

Case 3 (Round 3 of Protocol 3.1 has completed, but Round 1 of Protocol 3.2 has not finished.). Note that V is finalized at $\texttt{version} \in \{1, 2, 3\}$, and therefore with outcome $\{A : a, B : b\}$, where $a + b = a_0 + b_0$ by Lemma 3.1.

Since P_i signed V's post-fund setup state, Round 3 and therefore Round 2 of Protocol 3.1 has completed. This implies the latest supported state held by P_i for both L_i and L_{i-1} include a guarantee targeting V. Furthermore, as Round 1 of Protocol 3.2 has not terminated, these supported states have a greater version number than any other supported states held by anyone in their respective channels, implying that P_i can ensure that both L_i and L_{i-1} finalize with outcomes including the guarantee targeting V introduced in Round 2 of Protocol 3.1.

Let $o = \texttt{statusOf}(L_{i-1}).\texttt{outcome}$. Calling $\texttt{Reclaim}(L_{i-1},)$ mutates o on-chain, deleting $o[V]$, adding a to $o[P_{i-1}].\texttt{amount}$ and adding b to $o[P_i].\texttt{amount}$.

Similarly, P_i's balance in L_i increases by b, implying P_i's balance increases by $a + b = a_0 + b_0$ across both L_i and L_{i-1}.

Case 4 (Round 1 of Protocol 3.2 has finished.). In this case, V is finalized at $\texttt{version} = 4$. The analysis is identical to Case 2.

□

Acknowledgement

Thanks to Julie Poole and Martin Slagorsky for clarification on terminology.

References

[1] Magmo, "nitro-protocol/gas-benchmarks/gas.ts," 2021. `https://github.com/statechannels/statechannels/blob/000e7de/packages/nitro-protocol/gas-benchmarks/gas.ts#L72-L110`.

[2] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016. `https://www.bitcoinlightning.com/bitcoin-lightning-network-whitepaper/`.

[3] J. Coleman, L. Horne, and L. Xuanji, "Counterfactual: Generalized state channels," *Acessed: Nov*, vol. 4, p. 2019, 2018. `https://l4.ventures/papers/statechannels.pdf`.

[4] T. Close, "Nitro protocol.," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 219, 2019. `https://ia.cr/2019/219`.

[5] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment channels over cryptographic currencies.," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 635, 2017. `https://ia.cr/2017/635`.

[6] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), pp. 949–966, Association for Computing Machinery, 2018.

[7] S. C. contributors, "Introducing web3torrent." blog, June 2020. `https://blog.statechannels.org/introducing-web3torrent/`.

[8] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party virtual state channels," in *Advances in Cryptology –EUROCRYPT 2019* (Y. Ishai and V. Rijmen, eds.), (Cham), pp. 625–656, Springer International Publishing, 2019. `https://ia.cr/2019/571`.

[9] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei, "Donner: Utxo-based virtual channels across multiple hops." Cryptology ePrint Archive, Report 2021/855, 2021. `https://ia.cr/2021/855`.

[10] C. Newcombe *et al.*, "Use of formal methods at amazon web services," 2021. `https://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf`.

[11] T. Close and A. Stewart, "Breaking state channels with tla+," 2020. `https://blog.statechannels.org/breaking-state-channels/`.

[12] A. Stewart, “Scalable multi-hop payments for the filecoin retrieval market,” 2021. `https://github.com/filecoin-project/devgrants/issues/348`.

A Virtual channel rules specification

In this section, we give a specification of some simplified ”virtual channel” application logic for the virtual channel V in Protocol 3.1. Once V is funded, Alice and Bob wish to make payments between each other without involving any intermediaries.

This is achieved by implementing `latestSupportedState` rules for V that allow Alice and Bob to “transition” to new states, changing V’s outcome based on some privately exchanged information:

- The setup states are given type `None`.
- Alice can move from a state with type `None` to a state with type `A`, or a state with type `B` to a state with type `AB`.
- Bob can move from a state with type `None` to a state with type `B`, or a state with type `A` to a state with type `AB`.

We describe a simple virtual channel app, where Alice & Bob can co-sign “vouchers” that can be used to update V’s outcome. Vouchers are a placeholder for state updates of a generalized state channel. In this sense, vouchers emulate a simple, embedded, two-party payment channel between Alice and Bob in V. Each of Alice & Bob have a single opportunity to present the latest doubly-signed voucher, ensuring both end-users have the opportunity to record the status-quo on-chain. More general virtual channel rules enable Alice and Bob to use generalized application-specific logic – prototype solidity code may be found in the `statechannels/statechannels` Github repository.[16]

```
interface Voucher {
  virtualChannelId: bytes32,
  version:          uint,
  alice:            uint,
  bob:              uint,
  // signatures on {virtualChannelId, version, alice, bob}
  signatures: Signature[],
}

function requireConsensus({consts, vars}: SupportProof) {
    require(vars.length == 1)
    require(signedByAll(vars[0], consts.peers))
}

function latestSupportedState(proof: SupportProof)) {
  let {consts, vars} = proof

```

[16]This code implements a `validTransition` function whose logic may be used in the implementation of a `latestSupportedState` function.

```
  if (vars.at(-1).version <= 1) {
    // version 0 is used for pre fund setup
    // version 1 is used for post fund setup
    requireConsensus(proof)
  } else if (vars.last.version == 4) {
    // version 4 is used during concluding
    requireConsensus(proof)
    require(vars[0].isFinal)
  } else {
    validateOutcomeChange(proof);
  }

  return {consts, vars: vars.at(-1)} // the last state received
}

function validateOutcomeChange({consts, vars}) {
  // Allowed named-state transitions are
  //     AB
  //     ^^
  //    /  \
  // A       B
  // ^       ^
  //  \     /
  //    None

  alice = consts.peers[0]
  bob = consts.peers.at(-1)

  require(vars.length in [1,2,3])
  let (start, challenge, response) = vars

  require(start.type == None)
  require(signedByAll(consts, start))

  if challenge is undefined:
    return start

  require(start.version = 1)

  require(challenge.version == 2)
  if challenge.type == A {
    require(signedBy(consts, challenge, alice))
  } else {
    require(challenge.type == B)
    require(signedBy(consts, challenge, bob))
  }


  let voucher1 = challenge.appData
  require(voucherSignedByAll(voucher1, [alice, bob]))

  let x = start.outcome[0].amount + start.outcome[1].amount

  // The following two requirements guarantee that intermediary's
  // funds are invariant
  require(voucher1.alice + voucher1.bob == x)
  require(challenge.outcome == {
    alice: voucher1.alice,
    bob: voucher1.bob
  })

  if response is undefined:
    return challenge

  require(response.version == 3)
  require(response.type == AB)
  if challenge.type == A {
    require(signedBy(consts, response, bob))
  } else{
    require(signedBy(consts, response, alice))
  }

  let voucher2 = response.appData
  require(voucherSignedByAll(voucher2, [alice, bob]))
  require(voucher2.version > voucher1.version)

  require(voucher2.alice + voucher2.bob == x)
  require(response.outcome == {
    alice: voucher2.alice,
    bob: voucher2.bob
  })

  return response
}
```

Listing 3: *Virtual Channel Rules*

These rules are designed to provide the following lemma.

Lemma A.1. *Suppose V finalizes on-chain.*

Either V is finalized at turn 0 with its original outcome $\{A : a_0, B : b_0\}$ or every party has signed a `None` *state with version 1, in which case peers can take unilateral actions to ensure V finalized with an outcome $\{A : a, B : b\}$, where $a_0 + b_0 = a + b$.*

Specifically, intermediaries can ensure that $o[A] + o[B]$ are invariant by only committing to a state with `isFinal` $=$ `true` *when it doesn't change the value of $o[A] + o[B]$.*

These rules miss some edge cases. Notably, Alice may submit a challenge while a signature is in flight from Bob to Alice. We take a pragmatic stance in this case, recognizing the possibility of addressing such edge cases with more complex logic.

B Future work

B.1 Full specification

This work glosses over depositing into and withdrawing from a channel. There are design choices when entering and exiting a channel which have ramifications on UX as well as gas costs.

We will write a full specification of SATP including deposits and withdrawals.

B.2 Formal verification

Most results in the theory of state channels use the Universal Composability (UC) framework to prove security guarantees.

We plan to explore formalizing the main results of this paper, and are considering theoretical frameworks such as UC, as well as more tangible techniques, such as TLA+.

B.3 Partial top-ups and checkouts

An in-use virtual channel V between Alice and Bob may end up with all the funds "at one end". If Alice runs out of funds, it may be smoother to top up V rather than fund a new V'.

B.4 Reduced latency of construction.

There are modifications to the code in A which enable a 2-round version of Protocol 3.1. The rough idea can be found in a draft version of this spec.