Payment Channels Overview

Orfeas Stefanos Thyfronitis Litos

University of Edinburgh
o.thyfronitis@ed.ac.uk

Abstract. This is an overview of the existing literature on virtual payment channels. Lightning [1], Perun [2] and TeeChan [3] are considered.

1 Introduction

Payment channels are constructions that permit the secure exchange of assets between remote agents without the need for each transaction to be recorded in a global database. They are constructed in a way that gives the opportunity to the cheated agents to report the latest valid state to a global database (i.e. blockchain) and reclaim their assets.

For example, imagine that *Alice* works in *Bob*'s pin factory. They have agreed that *Alice* be paid right after she makes each pin a small amount x [4]. This can add up to hundreds, even thousands small of payments each day. Since most cryptocurrencies impose fees per transaction, it would be a waste to broadcast a new transaction for each small payment. For this reason, they turn to payment channels.

At the beginning of each month, *Bob* creates a transaction that pays e.g. 100 coins (a bit more than *Alice*'s expected pay for the month) to himself. He builds it in a way that needs both his and *Alice*'s signature to be spent (i.e. 2-of-2 multisig). This is the "bond" transaction. *Alice* confirms that the "bond" looks fine and gives *Bob* a special transaction that spends the "bond". This transaction is the "refund" transaction. *Bob* broadcasts the "bond" (but not yet the "refund") to the blockchain. The channel is now open.

Every time *Alice* makes a pin, *Bob* pays x to *Alice* as follows: He creates a new "refund" that pays to *Alice* the amount she already owned according to the previous "refund" plus x; accordingly, his payment is reduced by x. The total coins in the refund are always the same. He signs the new "refund" and sends it to *Alice*. She in turn signs the new "refund" and sends it back to *Bob*. The channel is now updated.

Finally, the end of the month comes and *Alice* wants to cash out on the blockchain, so that she can use her coins elsewhere. In order to do so, she simply broadcasts the latest "refund". The "bond" is spent according to the latest update, so she takes her rightful payment and *Bob* takes the rest of the 100 initial coins. The channel is now closed.

Note that exactly two transactions have been broadcast on the blockchain no matter how many payments were made, so the fees are kept low. Furthermore, both parties can unilateraly close the channel at any given point and claim the coins of their latest "refund", thus no trust is required between the two parties.

As an extension of the previous model, let *Charlie* be a colleague of *Alice*, who also has a payment channel with *Bob*. It is reasonable to imagine a system where *Alice* can pay *Charlie* without touching the blockchain, by leveraging the two pre-existing channels ($Alice \Leftrightarrow Bob, Bob \Leftrightarrow Charlie$) with minimal interaction with *Bob* and without having to trust him at all.

In the following sections we will summarise and compare various specific constructions that realise the high-level ideas described above. We will use the original terminology used in each paper.

2 Lightning Network

This construction is the first to achieve a functional model for payment channels. It is designed for bitcoin and requires some new opcodes and removing the malleability of transactions to function properly [1].

2.1 Simple two-party channel

The basic construction is as follows. Suppose that *Alice* and *Bob* want to create a payment channel that contains 1 BTC consisting of 0.5 BTC from each party. To achieve this, they follow these steps (see also section 3.1.2 and Figure 4 in 3.3.2 in [1]):

1. Either party (say *Alice*) creates a "Funding" transaction (F) with an input of 0.5 BTC from her and 0.5 BTC from *Bob*, and a 2-of-$\{Alice, Bob\}$ multisig as output; she then sends F to *Bob*. This transaction is not yet signed nor broadcast. F needs to be signed by both parties to be valid.
2. *Alice* creates, signs and sends to *Bob* a "Commitment" transaction ($C1b$) that spends F and has the following outputs:
   (a) 0.5 BTC that can be spent by *Alice* immediately when $C1b$ is broadcast.

(b) 0.5 BTC that can be spent by either party, but *Bob* can spend it only after a specified amount of blocks (say n) have been mined on top of $C1b$, whereas *Alice* can spend it only if *Bob* provides her with a "Breach Remedy" transaction (explained later) signed by him. This output is called "Revocable Sequence Maturity Contract" (RSMC).

Furthermore, *Alice* creates, signs and sends a "Revocable Delivery" transaction ($RD1b$) that pays the first of the two outputs of $C1b$ to *Bob*, but will be accepted by the network if it is in the mempool only after n blocks have been mined on top of $C1b$.

Bob similarly creates, signs and sends $C1a$ and $RD1a$ to *Alice*.

3. After *Alice* receives the signed $C1a$ and $RD1a$ from *Bob*, she verifies that they are both valid and correctly spend F. Given that everything works out right, she signs F and sends it to *Bob*.

   Bob similarly verifies that $C1b$ and $RD1b$ have the correct structure, along with *Alice*'s signature on F. He then signs F and broadcasts it. Note that he does not have to trust *Alice* in any way.

   The fact that *Alice* holds $C1a$ and $RD1a$, already signed by *Bob*, ensures her that her 0.5 BTC cannot be locked in the 2-of-2 multisig of F in case *Bob* stops cooperating. If she decides that *Bob* stopped cooperating, she can broadcast $C1a$, wait for it to be confirmed n times and broadcast $RD1a$ to get her money back. Thus *Alice* need not trust *Bob* either.

   Observe that if *Bob* refuses to cooperate in signing F, then the blockchain has not been changed and no funds are at risk. In such case, to ensure that *Bob* cannot lock her funds in the future, she should immediately transfer her funds to a new address or periodically check the blockchain for F and broadcast $C1a$ and $RD1a$ in case she finds F on the ledger.

After initially setting up the channel, *Alice* and *Bob* can update it as follows (see also section 3.3.4 and Figures 7, 8 in [1]):

1. Both *Alice* and *Bob* follow exactly the same steps as before to create $C2a$, $C2b$, $RD2a$ and $RD2b$; the only difference these transactions have to their counterparts from the previous state of the channel is that, instead of 0.5 BTC for each player, they contain the new agreed balance of the channel (e.g. 0.4 BTC for *Alice* and 0.6 BTC for *Bob*).
2. *Alice* creates, signs and sends to *Bob* a so-called "Breach Remedy" transaction ($BR1a$). This transaction lets *Bob* redeem the RSMC output of $C1a$ as soon as $C1a$ is broadcast. *Bob* similarly creates, signs and sends $BR1b$ to *Alice*.

Note that this effectively disincentivises *Alice* from ever broadcasting $C1a$, since in such case *Bob* will have a window of n blocks during which he can claim the entire sum in $C1a$, 1 BTC, for himself. *Alice* had better purge $C1a$ after $BR1a$ is sent to *Bob*. Similarly *Bob* is incentivised to refrain from ever broadcasting $C1b$.

This arrangement creates a situation where both players can be confident that the state of the channel is the one expressed by $C2a$, $C2b$, $RD2a$ and $RD2b$, thus they can assume that *Alice* has just paid *Bob* 0.1 BTC. No trust between the two players was needed all along. There are only two caveats: First, both players must periodically check the blockchain to ensure that the other party has not broadcast an old Commitment transaction. Second, in case of an uncooperative counterparty, one has to wait a prespecified amount of time before releasing their funds, which may be undesirable.

Thus, the necessary number of blocks mined on top of a Confirmation transaction for a subsequent Revocable Delivery to be valid (previously called n) must be carefully chosen in a way that does not lock up the funds for a long time in case of a dispute and at the same time does not require that the parties check the blockchain too often for a malicious broadcast of an already invalidated Commitment transaction.

Alice can outsource the task of the periodic check to a dedicated service by sending it all the previous Breach Remedy transactions. To incentivise the service to cooperate, *Alice* can pay a fee to it as an output of these transactions. Note that *Alice* does not need to trust the service, since the only thing it can do is to broadcast a Branch Remedy transaction that was created by *Alice*; she never discloses any of her private keys to it.

Finally, the parties can cooperatively close the channel without having to wait n blocks as follows: When both parties have agreed to closing the channel, *Alice* creates, signs and sends to *Bob* an "Exercise Settlement" transaction (ES) that spends the Funding transaction and has two simple outputs, each paying to the respective party the sum of the last agreed Commitment transaction. Following the previous example, this transaction would pay 0.4 BTC to *Alice* and 0.6 BTC to *Bob*. *Bob* can then also sign and broadcast the transaction to close the channel.

Once *Alice* has sent ES, she considers the channel as closed. If *Bob* does not broadcast ES, we have a dispute and she has to broadcast the latest Commitment transaction and wait for her funds to be unlocked.

2.2 Payments depending on preimage knowledge (HTLC)

Multi-hop payments can take place between players (e.g. *Alice* and *Dave*) who do not share a simple channel (i.e. an on-chain Funding transaction), but share simple channels with intermediate nodes (e.g. *Alice* with *Bob*, *Bob* with *Carol* and *Carol* with *Dave*).

To enable the creation of multi-hop channels, so-called "Hashed Timelock Contracts" (HTLC) are used. An HTLC is an additional output in a Commitment transaction which can be redeemed by either *Alice* or *Bob*; *Alice* can redeem it after a specified number of additional blocks, say m, have been mined after the creation (*not* the broadcast) of the Commitment transaction, whereas *Bob* can redeem it at any time, but only if he produces the preimage R of a hash specified in the HTLC output (see also section 4.2 and Figure 12 in [1]).

More specifically, consider $C2a, C2b$ where, contrary to the example in the previous subsection, *Alice* has paid the 0.1 BTC to an HTLC instead of directly to *Bob*. *Bob* should be able to redeem the 0.1 BTC only if he knows the preimage R before the m blocks have been mined. In addition to $RD2a$ and $RD2b$, six additional transactions have to be signed and exchanged.

1. *Alice* signs and sends an "HTLC Execution Delivery" transaction ($HED1a$) to *Bob*. $HED1a$ pays the HTLC output of $C2a$ to *Bob*, only if he knows the required preimage R. Only *Bob* can broadcast the transaction.
2. *Bob* signs and sends a so-called "HTLC Timeout Delivery" transaction ($HTD1b$) to *Alice*. $HTD1b$ pays the HTLC output of $C2b$ to *Alice*, only after m blocks have been mined from the time $C2b$ was created. Only *Alice* can broadcast this transaction.
3. *Alice* signs and sends an "HTLC Execution" transaction ($HE1b$) to *Bob*. $HE1b$ pays the HTLC output of $C2b$ to *Bob*, only if he knows the required preimage R. Only *Bob* can broadcast this transaction. Its single output is an RSMC with duration n, spendable by *Bob*.
4. *Alice* signs and sends an "HTLC Execution Revocable Delivery" transaction ($HERD1b$) to *Bob*. This transaction spends the RSMC output of $HE1b$. *Bob* can broadcast this transaction after n blocks have been mined on top of $HE1b$.
5. *Bob* signs and sends an "HTLC Timeout" transaction ($HT1a$) to *Alice*. $HT1a$ pays the HTLC output of $C2a$ to *Alice*, only after m blocks have been mined from the time $HT1a$ was created. Its single output is an RSMC with duration n, spendable by *Alice*.

6. *Bob* signs and sends an "HTLC Timeout Revocable Delivery" transaction ($HTRD1b$) to *Alice*. This transaction spends the RSMC output of $HT1b$. *Alice* can broadcast this transaction after n blocks have been mined on top of $HE1b$.

Note that once again, no trust is necessary in the process described above. The RSMC outputs of $HT1a$ and $HE1b$ are necessary for future invalidation according to the "Breach Remedy" method. More details can be found in Figure 14 of section 4.3. In case of common desire to close the channel, they can be cooperatively closed using the "Exercise Settlement" method.

2.3 Multi-hop channels

With the use of HTLC outputs, it is possible to execute multi-hop payments as follows. Suppose *Alice* wants to pay *Dave* 0.001 BTC and they find out that they are connected through the preexisting channels *Alice* ⇔ *Bob*, *Bob* ⇔ *Carol* and *Carol* ⇔ *Dave*. This payment can be completed with the following steps:

1. *Dave* generates a random number R and sends $hash\,(R)$ to *Alice*, *Bob* and *Carol*.
2. *Alice* and *Bob* update their channel with an e.g. 300-block HTLC that transfers 0.001 BTC from *Alice* to *Bob*.
3. *Bob* and *Carol* update their channel with an e.g. 200-block HTLC that transfers 0.001 BTC from *Bob* to *Carol*.
4. *Carol* and *Dave* update their channel with an e.g. 100-block HTLC that transfers 0.001 BTC from *Carol* to *Dave*.
5. *Dave* discloses R to *Carol*; he obtains 0.001 BTC from the 100-block HTLC transaction.
6. *Carol* discloses R to *Bob*; she obtains 0.001 BTC from the 200-block HTLC transaction.
7. *Bob* discloses R to *Alice*; he obtains 0.001 BTC from the 300-block HTLC transaction.

Thus *Alice* has paid *Dave* 0.001. No party can be defrauded: For example, *Carol* will pay 0.001 BTC to *Dave* if he shows her R within 100 blocks but then she can take the 0.001 BTC back by disclosing R to *Bob*; she has at least 100 more blocks to do so. In case *Dave* does not disclose R, all parties can take their funds back by settling on-chain.

On the other hand, assume that *Bob* does not cooperate after the establishment of the HTLC transactions, but keeps R hidden. In this case *Bob* will lose his 0.001 BTC to *Carol* and no other player will be negatively affected; *Carol* and *Dave* can fulfill their part without *Bob*'s cooperation, albeit *Carol* will have to wait for her channel with *Bob* to expire, since she has to settle on-chain. Likewise *Alice* can take back her 0.001 after the 300-block HTLC lock has expired. Thus no trust between parties is needed.

One can note three things: Firstly, there is no such thing as a persistent multi-hop channel. The whole procedure must be repeated for each subsequent multi-hop payment and the successful completion of one such payment does not facilitate the creation of future payments along the same route as far as the techniques described above are concerned. Nevertheless, previous cooperation between players can obviate the need of exploring the network anew for a connecting series of preexisting channels.

Secondly, merely the existence of a channel is not enough to ensure that multi-hop payments can be achieved through it. It must be the case that the correct player holds at least as much funds as the desired payment, which can only be verified by asking the players of the channel, since the latest state is not public. Thus, in the previous example, *Bob* must own at least 0.001 BTC in the *Bob* $\Leftrightarrow$ *Charlie* channel in order for the payment to be possible. *Alice* (or *Dave*) must ask *Bob* and *Charlie* whether this is the case before initiating the multi-hop payment process.

Finally, all intermediate players have to actively engage for a multi-hop payment to go through. This means that a multi-hop payment's latency increases linearly with the length of the chain, as well as the waiting time if on-chain settlement is needed (given that the same margin of security is desired irrespective of the payment length). This reduces the scalability of the design and fosters the creation of centralized, heavily connected players that ensure that short chains are available instead of distributed, loosely connected players that exchange funds through long chains.

3 Perun

Perun [2] is a payment network designed for Turing-complete smart contract scripting languages. It has been implemented for Ethereum. Its main contribution is *multistate channels* that allow the dynamic deployment of virtual contracts, known as *nanocontracts*. Contracts of this type do not have to enter the blockchain if all parties are cooperative and only do so in case of a dispute.

The paper describes specifically the use of such multistate channels for creating virtual payment channels between parties that do not have a basic payment channel between them, but both have basic multistate channels with an intermediary. Then the intermediary could substitute for the blockchain and thus a virtual payment channel on top of the two basic multistate channels can be created. The parties need the intermediary only for setting up the channel and to close it fast. If the intermediary refuses to close the channel, they can always fall back to the blockchain in order to close it.

3.1 Basic payment channels

A basic payment channel is a tuple

$$\gamma = (\gamma.\mathtt{id}, \gamma.\mathtt{Alice}, \gamma.\mathtt{Bob}, \gamma.\mathtt{cash}, \gamma.\mathtt{ver\text{-}num}, \gamma.\mathtt{sign})$$

Versions of this tuple are held by *Alice* and *Bob*. $\gamma.\mathtt{id}$ is a unique identifier for the channel, $\gamma.\mathtt{Alice}$ and $\gamma.\mathtt{Bob}$ are the end-users of γ and $\gamma.\mathtt{cash}$ is a function from the end-users to a real non-negative value that denotes the amount of cash the user has in the channel. $\gamma.\mathtt{ver\text{-}num}$ is a number that is incremented with each channel update (so that the latest state of the channel is known in case of dispute) and $\gamma.\mathtt{sign}$ is the singature of the other party on $(\gamma.\mathtt{id}, \gamma.\mathtt{cash}, \gamma.\mathtt{ver\text{-}num})$.

A payment channel has a corresponding $\mathtt{PaymentContract}_{\gamma.\mathtt{id}}$ on the ledger. End-users interact with the contract only to set up and close the channel, whereas updating the channel happens off-chain. The contract does not contain the fields $\gamma.\mathtt{ver\text{-}num}$ and $\gamma.\mathtt{sign}$; the two fields are kept only by the end-users.

Channel creation

The procedure of creating a channel is as follows:

1. *Alice* creates a $\mathtt{PaymentContract}\,(\gamma)$, pays it $\gamma.\mathtt{cash}\,(\gamma.\mathtt{Alice})$ coins and broadcasts it on the ledger. The fields $\gamma.\mathtt{ver\text{-}num}$ and $\gamma.\mathtt{sign}$ are not included.
2. The contract sends the message ($\mathtt{initialising}$, γ) to both end-users ($\gamma.\mathtt{Alice}$ and $\gamma.\mathtt{Bob}$).
3. *Bob* calls the $\mathtt{confirm}()$ function of the contract and pays it the already specified amount of $\gamma.\mathtt{cash}\,(\gamma.\mathtt{Bob})$ coins.
4. The contract sends the message ($\mathtt{initialised}$, γ) to both end-users.

5. If *Alice* does not receive ($\texttt{initialised}$, γ) after a predefined period Δ has passed from receiving ($\texttt{initialising}$, γ), she calls the contract function $\texttt{refund}()$ and gets her deposit back.

Note that *Alice* can get her money back if *Bob* does not cooperate and *Bob* only pays the contract after he verifies that *Alice* has set up everything correctly. The contract code is public and thus end-users do not engage with it if it does not correspond to the expected code; no trust towards the contract is needed.

Channel update

Assume that the end-users want to update an existing channel balance from $\gamma.\texttt{cash}$ to $\texttt{cash}'$, where the total channel balance has remained unchanged:

$$\gamma.\texttt{cash}\,(\gamma.\texttt{Alice}) + \gamma.\texttt{cash}\,(\gamma.\texttt{Bob}) = \texttt{cash}'\,(\gamma.\texttt{Alice}) + \texttt{cash}'\,(\gamma.\texttt{Bob})$$

The procedure of updating to the new balance is as follows:

1. *Alice* builds a new channel tuple γ^{Alice} where
   - the fields $\texttt{id}$ and $\texttt{users}$ are as in γ,
   - $\gamma^{Alice}.\texttt{cash} = \texttt{cash}'$,
   - $\gamma^{Alice}.\texttt{ver-num} = \gamma.\texttt{ver-num} + 1$ and
   - $\gamma^{Alice}.\texttt{sign}$ is *Alice*'s signature on $\left(\gamma^{Alice}.\texttt{id}, \gamma^{Alice}.\texttt{cash}, \gamma^{Alice}.\texttt{ver-num}\right)$.
2. *Alice* sends γ^{Alice} to *Bob* and waits for his response.
3. *Bob* checks that all fields are as expected and replaces the old channel tuple, γ, with the newly tuple, γ^{Alice}. From his point of view, the payment has gone through.
4. *Bob* sends to *Alice* the updated channel, γ^{Bob}, of which all fields are the same as γ^{Alice} except for $\gamma^{Bob}.\texttt{sign}$, which is *Bob*'s signature on $\left(\gamma^{Bob}.\texttt{id}, \gamma^{Bob}.\texttt{cash}, \gamma^{Bob}.\texttt{ver-num}\right)$.
5. If *Alice* receives the expected γ^{Bob}, she replaces the old channel tuple with γ^{Bob}. From her point of view, the payment has gone through.

The above description holds symmetrically if *Bob* initiates the channel update. If any player diverges from these steps, the other player can assume that the first has been corrupted and should close the channel immediately.

Note that after the first update, the channel tuples held by the two players are not the same, their only difference being in the signature field.

Strictly speaking, this means that the description of updating a channel above abuses the notation when it refers to γ as the common previous channel state.

Also note that the following scenario may arise: *Alice* sends the updated version of the channel along with her signature, but *Bob* does not reply. In this case, *Alice* wants to close the channel since *Bob* is assumed to be corrupt, but the latest state of which she has *Bob*'s signature is one version earlier than *Bob*'s latest state. The only way *Alice* can retrieve her funds is by broadcasting this older state. *Bob* can then broadcast his latest state, which supersedes *Alice*'s state. From the point of view of the blockchain, *Alice* has tried to close the channel with an older state.

Since there is a situation where the blockchain cannot say which player was corrupt, *Alice* cannot be punished for broadcasting an older state of the channel by losing all her funds in the channel. She should be entitled to her share, as defined by the latest channel state that has been broadcast. Thus the punishment scheme of Lightning cannot be applied here.

Closing the channel

Finally, we present the procedure of closing a channel.

1. *Alice* calls the function $\texttt{close}\left(\gamma^{Alice}\right)$ of $\texttt{PaymentContract}_{\gamma.\texttt{id}}$.
2. $\texttt{PaymentContract}_{\gamma.\texttt{id}}$ checks that γ^{Alice} is correctly formed and holds the same total balance as the initial channel recorded in the contract. If so, it accepts γ^{Alice} as the channel state. Additionally, *Bob* can call $\texttt{close}\,()$ at any time and either *Alice* or *Bob* can call $\texttt{finalize}\,()$ after time Δ has passed. If γ^{Alice} is does not pass the checks, the contract ignores the call.
3. If *Bob* disagrees with the channel state published by *Alice*, he calls $\texttt{close}\left(\gamma^{Bob}\right)$ of $\texttt{PaymentContract}_{\gamma.\texttt{id}}$.
4. Upon receiving a $\texttt{close}\left(\gamma^{Bob}\right)$ call from *Bob*, $\texttt{PaymentContract}_{\gamma.\texttt{id}}$ checks that γ^{Bob} is correctly formed, holds the same total balance as the initial channel recorded in the contract and additionally has a higher version number than γ^{Alice}. If so, it accepts γ^{Bob} as the channel state. Either *Alice* or *Bob* can still call $\texttt{finalize}\,()$ after time Δ from *Alice*'s original $\texttt{close}\,()$ call has passed. If γ^{Bob} does not pass the checks, the contract ignores the call.
5. After time Δ has passed, either end-user can call $\texttt{finalize}\,()$ of $\texttt{PaymentContract}_{\gamma.\texttt{id}}$.

6. Upon receiving a $\texttt{finalize}()$ call from either end-user, the contract $\texttt{PaymentContract}_{\gamma.\texttt{id}}$ checks that time Δ has passed since the original $\texttt{close}()$. If so, it sends $\texttt{closed}$ and $\gamma.\texttt{cash}(P)$ to each end-user P. If not, it ignores the $\texttt{finalize}()$ call.

The above closing sequence gives *Bob* a window of duration at least Δ to dispute the closing channel state reported by *Alice*.

Note that, in contrast to Lightning, there is no provision for cooperative closing of a channel, thus a delay of Δ must always be incurred between initiating a channel closure and getting access to the funds. The parameter Δ is decided by the parties when the channel is created and presents the same tradeoffs as the parameter n of Lightning.

3.2 Multistate channels

A basic multistate channel is a tuple

$$\gamma = (\gamma.\texttt{id}, \gamma.\texttt{Alice}, \gamma.\texttt{Bob}, \gamma.\texttt{cash}, \gamma.\texttt{nspace}) \quad ,$$

where $\gamma.\texttt{id}$, $\gamma.\texttt{Alice}$, $\gamma.\texttt{Bob}$ and $\gamma.\texttt{cash}$ are as in a payment channel and $\gamma.\texttt{nspace}$ is a set of nanocontracts, or *nanocontracts space*.

The multistate channel γ has a corresponding contract $\texttt{MSContract}_{\gamma.\texttt{id}}$ on the ledger. The end-users have to interact with this contract upon channel creation, channel closure and in case of dispute over the state of a nanocontract. Note that the end-users can create new nanocontracts, as well as cooperatively update them, without touching the ledger.

A nanocontract $\nu \in \gamma.\texttt{nspace}$ is a tuple

$$\nu = (\nu.\texttt{nid}, \nu.\texttt{blocked}, \nu.\texttt{storage}, \nu.\texttt{ver-num}, \nu.\texttt{sign}) \quad ,$$

where $\nu.\texttt{nid}$ is a globally unique identifier of the nanocontract, $\nu.\texttt{blocked}$ is a function from the end-users of the multistate channel to a real non-negative value that denotes the amount of cash the end-user has in the nanocontract and $\nu.\texttt{storage}$ contains the storage of the nanocontract. Like simple payment channels, the nanocontract with the highest $\nu.\texttt{ver-num}$ and a valid $\nu.\texttt{sign}$ will be accepted by the blockchain in case of registration of the state of the nanocontract on the ledger.

Nanocontract creation and update

The update mechanism for a nanocontract is similar to the update mechanism of a simple payment channel and thus will not be explained in detail. The only substantial differences are the following:

1. After *Alice* proposes a nanocontract update, *Bob* has time Υ to reply whether he agrees with this update or not. If he agrees the update goes through, else the state of the nanocontract is not updated (apart from increasing the version number). This is not considered a dispute, so (on-chain) nanocontract state registration does not need to take place.
2. In case of a sucessful update, the cash balance of both end-users in the underlying multistate channel ($\gamma.\mathtt{cash}\,(Alice)$ and $\gamma.\mathtt{cash}\,(Alice)$) are updated to reflect the fact that the nanocontract update has consumed or returned some funds to the end-users.

Each nanocontract has its own $\nu.\mathtt{ver\text{-}num}$ and $\nu.\mathtt{sign}$ field, so that several nanocontracts of the same multistate channel can be updated in parallel. Let ν' be the state of the nanocontract ν after an update. The only requirement is that

$$\nu'.\mathtt{blocked}\,(Alice) + \nu'.\mathtt{blocked}\,(Bob) \leq \\ \nu.\mathtt{blocked}\,(Alice) + \nu.\mathtt{blocked}\,(Bob) \quad .$$

This ensures that no two nanocontracts will together require more funds than are available in the multistate channel and thus that all nanocontracts can be updated in parallel. If it is the case that $\nu.\mathtt{blocked}\,(Alice)+\nu.\mathtt{blocked}\,(Bob) = 0$, we say that the nanocontract ν is $\mathtt{terminated}$.

To create a new nanocontract ν, users simply apply the update mechanism. They have to ensure that $\nu.\mathtt{nid}$ is a new, globally unique identifier and that the $\nu.\mathtt{ver\text{-}num} = 0$.

Nanocontract registration

Registration of the state of a nanocontract on `MSContract` happens in case of dispute with regard to the state of the nanocontract or when the parties want to close the multistate channel. The registration mechanism is very similar to that of closing a simple payment channel, so will not be described in detail.

Given that a nanocontract ν is registered on the `MSContract` of the underlying multistate channel of ν, any end-user can unilaterally execute a nanocontract function `fun` by calling the `MSContract` function $\mathtt{execute}\,(\nu.\mathtt{nid}, \mathtt{fun}, z)$. `MSContract` then updates the state of the nanocontract on the ledger and returns the output to the end-users.

Nanocontract termination

Finally, when the end-users wish to close the multistate channel, they have to update all nanocontracts such that they are `terminated`, register their state (or alternatively register their state and then execute functions on the ledger until they are `terminated`) and then initiate a procedure similar to the closing of basic payment channels, which gives the opportunity to both end-users to publish their latest version of all nanocontracts of the multistate channel. The nanocontract states with the highest version number are accepted by the ledger as valid. Each end-user receives coins equal to the initial coins they contributed to the multistate channel, amended by the changes introduced by the nanocontracts. These coins are now available to use with other users of the ledger.

It may be the case that some nanocontracts cannot be updated to a `terminated` state due to dispute between end-users or design problems of the nanocontract. The end-users can have special provision in `MSContract` for such cases to be able to kill such misbehaving nanocontracts and distribute the funds in a predefined manner. Such a mechanism is not explicitly specified.

3.3 Virtual payment channels

Virtual payment channels are channels created on top of suitable preexisting multistate channels that facilitate trustless funds exchange between parties that do not share an on-chain channel. Going into more detail, a virtual payment channel γ is a tuple:

$$(\gamma.\mathtt{id}, \gamma.\mathtt{Alice}, \gamma.\mathtt{Ingrid}, \gamma.\mathtt{Bob}, \gamma.\mathtt{cash}, \gamma.\mathtt{subchan},\\ \gamma.\mathtt{validity}, \gamma.\mathtt{ver\text{-}num}, \gamma.\mathtt{sign})$$

Let *Alice* have a multistate channel with *Ingrid* ($Alice \overset{\gamma_a}{\Leftrightarrow} Ingrid$); also let *Bob* have a multistate channel with *Ingrid* ($Bob \overset{\gamma_b}{\Leftrightarrow} Ingrid$).

Channel creation

To build γ, two nanocontracts ν_a and ν_b are created, each on the corresponding multistate channel. ν_a has $\gamma.\mathtt{cash}\,(Alice)$ blocked by *Alice* and $\gamma.\mathtt{cash}\,(Bob)$ blocked by *Ingrid*. Similarly, ν_b has $\gamma.\mathtt{cash}\,(Alice)$ blocked by *Ingrid* and $\gamma.\mathtt{cash}\,(Bob)$ blocked by *Bob*. At a high level, a virtual payment channel creation protocol is as follows:

0. *Alice* and *Bob* discover that *Ingrid* is an intermediary. They also agree on the initial balance of γ.

1. *Alice* sends a signed ν_a to *Ingrid*.
2. *Ingrid* sends a signed ν_b to *Bob*.
3. *Bob* replies to *Ingrid* with ν_b, signed by the former.
4. *Ingrid* replies to *Alice* with ν_a, signed by the former.

We now say that there exists the virtual payment channel γ between *Alice* and *Bob* (*Alice* $\overset{\gamma}{\leftrightarrow}$ *Bob*).

Channel update

Updating the cash balance of the channel can be accomplished in the same way as for the basic payment channels, with both *Alice* and *Bob* signing the new state with an incremented version number. Note that, in contrast to Lightning, the end-users do not need to interact with *Ingrid* at all to update the channel. This decreases the number of rounds needed for an update to 2 in the optimistic case that both *Alice* and *Bob* are honest. Furthermore, it somewhat increases the privacy of the end-users.

Closing the channel

In case all three parties are honest, they agree that they want to close γ and γ.`validity` time has not yet passed, then both *Alice* and *Bob* attempt to terminate their respective nanocontract with *Ingrid*, ν_a and ν_b. The end-users send their latest version of γ to the intermediary, who expects a tuple with valid signatures by the initially registered end-users and a total balance equal to that of the initial state of the channel. If both end-users send different valid tuples, then *Ingrid* chooses the one with the higher version number as valid. Thus both nanocontracts can be terminated with each of the end-users unblocking their respective balance, as defined by the valid γ, on their multistate channel with *Ingrid*. The sum of coins *Ingrid* will unblock in both multistate channels will be equal to the sum of the original coins she blocked during the virtual payment channel creation, only redistributed between the two channels as defined by the latest γ state.

Alice can unilaterally register the ν_a nanocontract state on the ledger and provide her latest γ version, thus she does not need *Ingrid*'s cooperation to unblock her funds. If *Ingrid* learns a newer valid γ version from *Bob* (which means that *Alice* tried to cheat), then *Ingrid* can publish it within a predetermined timeframe and claim her rightful funds.

Furthermore, if the channel hasn't closed after γ.`validity` time has passed, any of the three parties can unilaterally finalize the nanocontract(s) she has access to and unblock the respective funds.

Thus we have seen that no trust between the parties is necessary. Similarly to Lightning though, cooperating parties can unblock their funds faster and with less interaction with the ledger (and thus lower fees).

4 Sprites

Sprites [5] constitute an improvement upon Lightning [1] regarding the worst-case time needed to settle in case of a dispute. Consider a channel of l hops, where Δ is the time given to each participant to publish their state after a counterparty has unilaterally broadcast theirs. The worst-case time to settle in the case of Lightning is $\Theta(l\Delta)$, whereas in Sprites it is $\Theta(l + \Delta)$.

To achieve this, Sprites propose a smart contract called Preimage Manager (`PM`). Let $\mathcal{H}(\cdot)$ be a suitable hash function. Parties can interact with `PM` in the following way:

- Call `publish`(x) at time T: `PM` stores `timestamp`$[\mathcal{H}(x)] = T$.
- Call `published`(h, T): `PM` returns `True` if
  - $h \in$ `timestamp` and
  - `timestamp`$[h] \leq T$,

  `False` otherwise.

In case all parties are honest, `PM` is not invoked, the entire interaction happens off-chain and needs $l+1$ rounds to complete. In case a party misbehaves by delaying sending the preimage until the last possible moment (i.e. time Δ after she received the preimage from the previous link), she will have to publish the preimage to the blockchain instead of just sharing it with the next link in the chain of payments in order to ensure she gets her funds. Thus, the rest of the (honest) players can settle the channel by asking `PM` whether the hash they already know has been `published()`. This action can be completed concurrently, thus the maximum delay that can be incurred is $l + \Delta$.

5 General properties of Payment Channels

1. Number of participants in the channel
2. On-chain connection(s) between participants
3. Actions: open, update, execute, close

4. Who needs to sign for each action, who is notified, how many rounds of communication?
5. What information can one obtain by observing the blockchain?
6. Under what circumstances an operation cannot complete? (e.g. concurrency issues)
7. Which participants are aware of the identity of which participants?
8. Is there an upper bound to the amount of updates? How is this number decided?
9. Can a participant unilaterally commit on-chain?
10. Up to how much money can a participant unilaterally obtain?
11. What can a malicious party do? If it corrupts more participants it can do more?
12. Can a malicious/honest-but-curious party that is a participant learn who is transacting with who?
13. How much slower is the process in case of a malicious party?
14. How expensive are the actions? (CPU, memory, storage)
15. How expensive are interactions with the blockchain? (fees, time, etc.)

Π_{LN}

Initialisation:
blocked $\leftarrow 0$
$G \leftarrow (\{\mathrm{sid}_{Alice}\}, \emptyset)$

Upon receiving (open, sid_{Bob}, x) from $\mathcal{E}$:
if $\mathcal{G}_{\mathrm{Ledger}}$.balance($Alice$) $-$ blocked $\geq x$ **then**
    blocked $\leftarrow$ blocked $+ x$
    Let tx be a suitable funding transaction
    Send (opening, x, tx) to Bob
    Send (INPUT, tx) to $\mathcal{G}_{\mathrm{Ledger}}$
    Wait for tx to be confirmed ▷ correct way to Wait?
    blocked $\leftarrow$ blocked $- x$
    $G \leftarrow G \cup (\{\mathrm{sid}_{Bob}\}, \{(\mathcal{H}(\mathrm{tx}), (\mathrm{sid}_{Alice}, x), (\mathrm{sid}_{Bob}, 0))\})$ ▷ Add sid_{Bob} to nodes, new channel to edges
    Send (opened, $Alice$, Bob, x, 0, $\mathcal{H}(\mathrm{tx})$) to $\mathcal{E}$
end if

Upon receiving (opening, x, tx) from Bob:
Wait for tx to be confirmed
$G \leftarrow G \cup (\{\mathrm{sid}_{Bob}\}, \{(\mathcal{H}(\mathrm{tx}), (\mathrm{sid}_{Alice}, 0), (\mathrm{sid}_{Bob}, x))\})$ ▷ Add sid_{Bob} to nodes, new channel to edges

Send (opened, *Alice*, *Bob*, 0, x, $\mathcal{H}(\text{tx})$) to $\mathcal{E}$

Upon receiving (newChannel, txid, $\text{sid}_{Charlie}, \text{sid}_{Dave}$) from *Bob*:
if there exists a tx with $\mathcal{H}(\text{tx}) == \text{txid}$ on $\mathcal{G}_{\text{Ledger}}$ and is a valid funding transaction **then**
    $G \leftarrow G \cup (\{\text{sid}_{Charlie}, \text{sid}_{Dave}\}, \{(\text{txid}, (\text{sid}_{Charlie}, \perp), (\text{sid}_{Dave}, \perp))\})$ ▷ State of channel unknown, thus balances are $\perp$
end if

Upon receiving (closedChannel, txid) from *Bob*:
if there exists a tx with $\mathcal{H}(\text{tx}) == \text{txid}$ on $\mathcal{G}_{\text{Ledger}}$ that closes a funding tx' that corresponds to edge e in G **then**
    $G \leftarrow G \setminus \{e\}$
end if

Upon receiving (pay, *Bob*, x) from $\mathcal{E}$:
Send (SendInvoice, x) to *Bob*
Wait for response (invoice, x, hash) from *Bob*:
if there is an $(Alice, Charlie, \ldots, Bob)$ path in G where the first hop is of weight at least x **then**
    Send a Sphinx [6] message with the correct HTLCs (containing hash) for *Bob*
    ▷ Sane fees and timeouts as requested by each hop
    Wait for (preimage) from *Charlie*
    if $\mathcal{H}(\text{preimage}) == \text{hash}$ **then**
        Let $e = (\text{txid}, (\text{sid}_{Alice}, y), (\text{sid}_{Charlie}, z))$ *Alice*'s channel with *Charlie*
        Update e to pay *Charlie* x ▷ use HTLC primitive
        $G \leftarrow G \setminus \{e\}$
        $G \leftarrow G \cup \{(\text{txid}, (\text{sid}_{Alice}, y - x), (\text{sid}_{Charlie}, z + x))\}$
        Send (paymentSent, *Bob*, x) to $\mathcal{E}$
    else
        Send (paymentFailed, *Bob*, x) to $\mathcal{E}$
    end if
else
    Send (noPath, *Bob*, x) to $\mathcal{E}$
end if

Upon receiving (SendInvoice, x) from *Bob*:
preimage $\xleftarrow{r} \{0,1\}^{\text{gazillion}}$

hash $\leftarrow \mathcal{H}$ (preimage)
Send (invoice, x, hash) to *Bob*
Wait for update of any channel $e = (\text{txid}, (\text{sid}_{Alice}, y), (\text{sid}_{Charlie}, z))$ to $e' = (\text{txid}, (\text{sid}_{Alice}, y + x), (\text{sid}_{Charlie}, z - x))$ conditional on *Alice*'s knowledge of the preimage of the hash
Send (preimage) to *Charlie*
Wait for update of e' to $e'' = (\text{txid}, (\text{sid}_{Alice}, y + x), (\text{sid}_{Charlie}, z - x))$ unconditional
if *Charlie* does not update the channel to e'' **then**
    Settle e' on-chain with the preimage and take x from the HTLC provided by *Charlie*
end if
Send (paymentReceived, *Bob*, x) to $\mathcal{E}$

Upon receiving (close, id) from *Alice*:
if $\mathcal{G}_{Ledger}$ has a valid funding tx with $\mathcal{H}$ (tx) == id **then**
    Try to close cooperatively ▷ TODO
    if cooperative closing fails **then** ▷ TODO
        Close unilaterally ▷ TODO
    end if
end if

6 Notes on Lightning Specification

- The relevant part of the specification can be found at `https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md`.
- Currently only the opener (sender) of a channel funds it.

7 Model for Payment Channels

A payment channel is a tuple

$$PC = (\{(P_1, c_1), \ldots, (P_n, c_n)\}, \{(e_1, b_1), \ldots, (e_m, b_m)\}, f : \mathcal{A}^n \rightarrow \mathcal{PC})$$

where $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{m} b_i$.

(P_i, c_i) represents the i-th player and her available funds on settling.

(e_j, b_j) represents the j-th on-chain endpoint and the corresponding funds that will be released for use in the blockchain if this endpoint is settled.

Algorithm 1 $\mathcal{F}_{\text{PayNet}}$

Initialisation:
for all $v \in \mathcal{P}$ **do**
    blocked $(v) = 0$
end for
Pending $= \emptyset$
Channels $= \emptyset$

Upon receiving (open, Bob, x, y) from $Alice$:
if $(Bob, Alice, y, x) \in$ Pending **then**
    assert($\mathcal{G}_{\text{Ledger}}$.balance($Alice$) $\geq$ blocked $(Alice) + x$)
    assert($\mathcal{G}_{\text{Ledger}}$.balance($Bob$) $\geq$ blocked (Bob))
    blocked $(Bob) =$ blocked $(Bob) - y$
    id $\stackrel{r}{\leftarrow} \{0,1\}^{\text{gazillion}}$
    Channels $\leftarrow$ Channels $\cup \{(Alice, Bob, x, y, \text{id})\}$
    Pending $\leftarrow$ Pending $\setminus \{(Alice, Bob, x, y)\}$
    create funding tx with id for $Alice, Bob$ with initial balance x, y respectively
    send message (opened, $Alice, Bob, x, y,$ id) to $Alice$
    send message (opened, $Bob, Alice, y, x,$ id) to Bob
else
    Pending $\leftarrow$ Pending $\cup \{(Alice, Bob, x, y)\}$
    blocked $(Alice) \leftarrow$ blocked $(Alice) + x$
end if

Upon receiving (pay, Bob, x) from $Alice$:
if there is an $Alice \rightarrow Bob$ path of weight x **then**
    for all $(Carol, Dave, y, z, \text{id})$ channel in the path **do**
        Update channel to $(Carol, Dave, y - x, z + x, \text{id})$
    end for
    send message (payed, Bob, x) to $Alice$
    send message (payedFrom, $Alice, x$) to Bob
else
    send message (noPath, Bob, x) to $Alice$
end if

Upon receiving (close, id) from $Alice$:
if $\mathcal{G}_{Ledger}$ has a valid funding tx with id **then**
    close that channel
end if

f is a function from player actions to a new payment channel. The new payment channel must have at most as much funds as the old.

References

1. Poon J., Dryja T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments
2. Dziembowski S., Eckey L., Faust S., Malinowski D.: PERUN: Virtual Payment Channels over Cryptographic Currencies. IACR: Cryptology ePrint Archive (2017)
3. Lind J., Eyal I., Pietzuch P., Sirer E. G.: Teechan: Payment Channels Using Trusted Execution Environments. ArXiv preprint arXiv:1612.07766 (2016)
4. Kuzmenko I.: Bitcoin Developer Guide. https://bitcoin.org/en/developer-guide
5. Miller A., Bentov I., Kumaresan R., Cordi C., McCorry P.: Sprites and State Channels: Payment Networks that Go Faster than Lightning. ArXiv preprint arXiv:1702.05812 (2017)
6. Danezis G., Goldberg I.: Sphinx: A compact and provably secure mix format. In Security and Privacy, 2009 30th IEEE Symposium on: pp. 269–282: IEEE (2009)