PERUN: Virtual Payment Channels over Cryptographic Currencies*

Stefan Dziembowski[1], Lisa Eckey[2], Sebastian Faust[2], and Daniel Malinowski[1]

[1] University of Warsaw
[2] Ruhr Bochum University

Abstract. Payment channels emerged recently as an efficient method for performing cheap *micropayment* transactions in cryptographic currencies. In contrast to the traditional on-chain transactions, payment channels have the advantage that they allow nearly unlimited number of transactions between parties without involving the blockchain. In this work, we introduce *Perun*, a new system for payment and state channels over cryptographic currencies, that has several advantages over the existing proposals. In particular, Perun offers a new method for connecting channels that is more efficient than the existing technique of "routing transactions" over multiple channels. That is, in contrast to prominent existing solutions such as the *Lightning Network*, Perun does not require involvement of the intermediary over which payments are routed. To achieve this, Perun introduces a new technique that we call *channel virtualization*, which allows to build *virtual payment channels* over so-called *multistate channels*. Multistate channels are a new primitive that we introduce as an independent concept in this work and generalizes the notion of "state channels". Our schemes can work over any cryptocurrency that provides Turing-complete smart contracts. As a proof of concept, we implemented Perun in *Ethereum*.

* Research supported by the Polish National Science Centre grant 2014/13/B/ST6/03540 and by the Emmy Noether Program FA 1320/1-1 of the German Research Foundation (DFG).

Table of Contents

1 Introduction

Decentralized cryptocurrencies such as Bitcoin [20] or Ethereum [28] have gained great popularity over the last years. They enable pseudonymous online payments, cheap remittance and many other novel applications. Due to this flurry of applications decentralized cryptocurrencies process a steadily increasing number of transactions, e.g., Bitcoin currently handles around 250.000 transactions per day. The backbone [11] of these currencies is a technology called *blockchain* (or *ledger*), which can be viewed as append-only register on which the transactions are stored, and whose entire contents is publicly available, and is checked for consistency by the users that maintain the system — the so-called *miners*. As the name suggests, blockchain is a chain of data blocks (containing transactions) that are created by the system at some rate. Unfortunately, this approach faces an inherent scalability problem that can significantly hinder further adaption. Since each transaction that is processed by the network has to be stored on the blockchain, there is a fundamental limit on how many transactions can be processed per second. For instance, in Bitcoin with its 1MB block size, and a block creation-rate of approximately 10 minutes, the network is currently limited to process up to 7 transactions per second [8].

A natural solution to this problem is to increase the block size, or the block creation rate, but even with these changes it is unlikely that ledger-based cryptocurrencies can reach the efficiency of centralized payment systems, e.g., the VISA network processes during peak times 56,000 transactions per second [8]. Notice that already at the current transaction rate the ledger in Bitcoin grows by approximately 10 GB every four months – reaching over 100GB in February 2017. Because the miners need to store and verify the entire transaction history starting from the very first, so-called *genesis* block, it would not be possible to maintain such a system in the long run.

The scalability problem is drastically amplified with the emergence of nanotransactions, which is one of the expected "killer applications" when ledger-based currencies go mainstream [27]. Nanotransactions allow users to transfer very small amounts of money, typically less than 1 cent, and can enable many novel business models, e.g., fair sharing of WiFi connection, or devices paying to each other in the "Internet of Things". Besides the scalability issues of nanotransactions, there are also several other challenges that need to be addressed by ledger-based cryptocurrencies before they can handle massive volumes of transactions. First, in many settings nanotransactions have to be executed instantaneously (e.g., in the application of sharing the WiFi connection, or when a user wants to read an article on a news webpage); which is a problem, since confirmation of transactions in Bitcoin takes at least around 10 minutes[3]. Secondly, and more importantly, when miners process transactions they ask for fees. While currently fees are relatively low, they are expected to rise when millions of transactions compete for the scarce source of fast processing. Once these fees surpass the actual value assigned to a transaction, nanopayments become much less attractive – something which we also witness for traditional online payment systems via credit cards or PayPal. Both systems do not offer nanotransactions due their fee system, e.g., PayPal asks for at least 30 cents for each transaction.

An exciting proposal to address the above challenges are so-called *payment channels* [7], which allow two users to rapidly exchange money between each other without sending transactions to the ledger. This is achieved by keeping the massive bulk of transactions off-chain, and using the ledger only when parties involved in the payment channel disagree, or when they want to close the channel. As long as the parties are not in conflict, they can freely *update* the balance of the channel (i.e. transfer money between each other's accounts). Because off-chain transactions can always be settled on a ledger by the users involved, there is no incentive for the users to disagree, and hence honest behavior is enforced. In the optimistic case, when the two parties involved in the payment

[3] This is reduced in other cryptocurrencies such as Ethereum, or in Bitcoin via zero-confirmation transactions.

channel play honestly, and off-chain transactions never hit the ledger before the channel is closed, payment channels significantly reduce transaction fees, allow for instantaneous payments and limit the load put on the ledger.

Technically, payment channels are implemented using so-called *smart contracts*, which informally speaking are formal agreements written on the ledger and executed by the cryptocurrency itself (we give a short introduction to this concept in Sect. 3). In the simplest case every channel has a corresponding smart contract that is posted on the ledger when the channel is created. Such a contract is later used to resolve disputes between users about the channel's state, or simply to close the channel. There are several ways to design such channels, ranging from very basic "unidirectional" constructions such as the one described in the "Rapidly-adjusted (micro)payments to a pre-determined party" section of [6], up to fully multidirectional ones, like some of those described in [22].

The concept of payment channels has been extended in several directions. One of the most important extensions are the so-called *payment networks*, which enable users to route transactions via intermediary hubs, hence reducing the on-chain transaction load, since the number of channels opened on the blockchain is reduced. An example of such a network has been designed by Poon and Dryja over Bitcoin [22]. In their system the payments are routed over the network in the following way. Suppose two parties, Alice and Ingrid, established a payment channel (denote it: Alice $\Leftrightarrow$ Ingrid), and Ingrid also has a payment channel with Bob, denoted Ingrid $\Leftrightarrow$ Bob (but Alice and Bob do not have a payment channel between each other). Then Alice can perform a nanopayment, for x coins to Bob *via the intermediary* Ingrid. The main technique to achieve this in Bitcoin-like cryptocurrencies are the so-called *hash-locked transactions*, which guarantee the following: the transfer of x coins from Alice to Ingrid (over Alice $\Leftrightarrow$ Ingrid) will be performed *if and only if* the transfer of x coins from Ingrid to Bob (over Ingrid $\Leftrightarrow$ Bob) is performed. Note that using hash-locked transactions for routing payments require interaction with Ingrid for every payment (but it does *not* require touching the ledger, as long as everybody is behaving honestly). This can be generalized to longer chains of payments.

A further generalization of payment channels are the *state channels* [9, 1], which significantly enrich the functionality of payment channels (cf. Sect. 4.1 for more details). For a moment let us just say that the users of state channels can, besides payments, execute some smart contract C "inside of a channel" in an off-chain way (as long as the users do not enter into a dispute). Very informally, this is done by letting the channel's state contain, in addition to the financial balance, a string σ that describes the current state of C's *storage* (i.e. the values of all the contract's variables). As long as there is no conflict between the users of the state channel, they can freely update σ. However, once one of the parties starts to misbehave, the other one can post the latest version of σ on the ledger, and the ledger will finish the execution of C (and take care of all its financial consequences), starting from storage σ. In most of the cases, the parties will not need to do it, and all the execution can be handled by simply updating the state σ in a "peaceful way".

Hence, in some sense state channels provide a way to implement a "virtual 2-party ledger", in the following way: two parties that established a state channel can maintain a "simulated transaction ledger" between themselves and perform the ledger transactions on it without registering them on the *real* ledger. This happens as long as the parties do not enter into a conflict. The security of this solution comes from the fact that at any time any party can pass the current state of the channel to the real ledger.

1.1 Our contribution

Channel virtualization. Our main contribution is the development of a new technique for connecting channels, that we call "channel virtualization". Our method is an alternative to "payment

routing" used in existing payment channel schemes and does not rely on the hash-locked transactions. Its main advantage is that channels created via channel virtualization are *non-interactive*, in the sense that they minimize the need for interaction with the intermediaries, and in particular there is no need for routing individual payments over the intermediates. More precisely, suppose we are again in the situation as above, i.e. there exist channels between Alice and Ingrid and between Ingrid and Bob. Our technique allows Alice and Bob to establish a *virtual* payment channel (denoted Alice $\leftrightarrow$ Bob) with the help of Ingrid (but without touching the ledger). As long as everybody is honest, Alice and Bob need to interact with Ingrid only when the channel is created and when it is closed. Each individual transaction between Alice and Bob that goes via this channel does *not* require interacting with Ingrid! Our scheme is secure against arbitrary corruptions of Alice, Ingrid, and Bob (in particular: no assumption about the honesty of the intermediary Ingrid is needed). To distinguish such *virtual* channels from the ones that are created directly on the ledger (e.g., the channels Alice $\Leftrightarrow$ Ingrid and Ingrid $\Leftrightarrow$ Bob) we call the latter *basic*.

Our construction is based on the idea of applying the channel construction recursively, by building a payment channel "on top of" state channels. Recall that state channels provide a "simulated contract ledger" between the involved parties. Our observation is that contracts in a state channel can be used to construct payment channels in a way similar to how smart contracts on the ledger can be used to build payment channels over the standard ledger.

There are many details that need to be taken care of in order for this general idea to work. What is especially delicate is handling *parallelism*, i.e., ensuring that several virtual channels (with overlapping sets of users) can be opened, updated, and closed simultaneously. In particular, we introduce an extended version of standard state channels, that we call the "*multi*state channels". Defining and constructing them is a second main contribution of our work.

Multistate channels. The properties of "multistate channels" are tailored to the virtual channels that we construct, but we believe that they are of independent interest. Their first important feature is that they have a more fine-grained control over the process of updating the channel's state. Suppose Alice $\Leftrightarrow$ Ingrid is some basic multistate channel. Our system consists of a mechanism that allows Alice to propose an update of this channel to Ingrid additionally specifying time Υ that Ingrid has to "respond" to this proposal (by accepting or rejecting it). The decision of party Ingrid can depend on some external conditions, e.g., in case of the virtual channel creation procedure, it will depend on the behavior of Bob. But we believe that such a conditional update mechanism can also be useful in many other use-cases where state channels can be applied.

The second feature that a multistate channel has is that it consists of several "independent" contract storage states $\sigma_1, \ldots, \sigma_m$ (in contrast with a *single* storage state σ, in the standard state channels). The σ_i's correspond to contracts that can be created, updated, and executed in parallel (we will call such contracts the "nanocontracts", see Sect. 4.2). To readers familiar with state channels it may look like there is not that much difference between multistate channels and standard state channels, as in principle we can simply view $(\sigma_1, \ldots, \sigma_m)$ as a *single* storage of a contract that operates on every σ_i independently. The main problem with this simple approach comes from parallelism. Consider the following example. Let Alice and Bob be parties that have a multistate channel that consists of contract storage states $\sigma_1, \ldots, \sigma_m$. Now, suppose that in the same moment Alice proposes Bob to update each σ_i to some σ_i'. In the multistate channels Bob can agree only to *some* of these proposals. For example, he may agree to update only σ_1 (and not agree to other updates). On the other hand, if $(\sigma_1, \ldots, \sigma_m)$ is treated as a *single* state, then he does not easily have such flexibility. As it turns out, this feature is important for handling parallelism of virtual channels.

We implemented our system called *Perun*[4] in *Solidity*, which is one of the languages used in the Ethereum currency [24] (see Appx. A). The most natural application of Perun is to provide a very fast way to stream tiny payments. For example, consider the situation when a client Alice pays for using WiFi to some Internet provider Bob (and they both have basic channels with an intermediary Ingrid). The "routing payments" approach (from Lightning) puts some inherent limitations on the size of each packet of data for which the client pays, as each payment requires interaction with Ingrid, which introduces delays and puts heavy communication load on Ingrid. By using our approach Ingrid is involved in the communication between Alice and Bob only when the session starts and when it ends. Another natural application of our technique is the Internet of Things. Due to the cost pressure to reduce the power consumption in many situations these devices will be connected via some short-range communication technology (like Bluetooth or NFC), and will minimize the interaction with remote devices. Hence, routing every payment via a third party server may not be an option in such situations. Our technique removes the need for such interaction. Another, related scenario where our solution can be applied is when the payment intermediary cannot be assumed to be always available. For example imagine payments in the vehicular ad hoc networks — here the permanent availability of the internet connections cannot be guaranteed (due to conditions like entering a tunnel, or a zone with no mobile phone network access).

In the future we plan to extend Perun by adding an option to create longer channels, and to create virtual *state* channels, which will allow their users to engage into contracts (and, e.g., play lotteries, or card games). Such extension are a subject of our subsequent work (see Sect. 4.4).

Let us also mention that the virtual channels provide a higher degree of privacy than the standard channels of [22], as the intermediary does not get information about the history of transfers between the parties. We would like to stress, however, that privacy in the payment channels is not the focus of this work (for more on this topic, see, e.g, [12, 13]).

Formal aspects. The focus of this work is an introduction of a new technique. Full formalization of our protocols, and complete proofs in the UC-framework are beyond its scope (handling formally all the security features in the fully concurrent settings requires several pages of definitions, not to mention the proofs). Such formalization appears in a separate work, where we also show an extension of this paper to longer virtual *state* channels.

1.2 Notation

Throughout this paper we will assume existence of a a signature scheme $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$ that is existentially unforgeable under an adaptive chosen-message attack (see, e.g., [15], Chapter 12). For two strings $a, b \in \{0,1\}^*$ the term $a||b$ denotes their concatenation. A function $f : \mathcal{A} \to \mathcal{B}$ is called *partial* if for some values $a \in \mathcal{A}$ we have $f(a) = \perp$ (i.e. $f(a)$ is "undefined"). An empty string will be denoted with ε.

We will assume that all values like numbers, functions, pairs of values, etc. are implicitly encoded as binary strings (e.g. when they are sent as messages). We will also use *keywords*, which are written using sans serif font ("$\mathsf{keyword}$), and (formally) represent some fixed binary strings. We will frequently present tuples of values using the following convention. The individual values in a tuple T are identified using keywords called *attributes*: $\mathsf{attr1}, \mathsf{attr2}, \ldots$. Formally an *attribute tuple* is a function from its set of attributes to $\{0,1\}^*$. The *value of an attribute* attr *in a tuple* T (i.e. $T(\mathsf{attr})$) will be referred to as $T.\mathsf{attr}$. This convention will allow us to easily handle tuples that have dynamically

[4] Perun is the god of thunder and lightning in the Slavic mythology. This choice of a name reflects the fact that one of our main inspirations is the Lightning system. The name "Perun" also connotes with "peer" (which reflects its peer-to-peer nature), and "run" (which stresses the fact that the system is very fast).

changing sets of attributes. For example when we say that "we add an attribute attr to T and set it to x" it means that T is replaced by T' with and additional attribute attr and $T'.\mathit{attr} = x$.

1.3 Further related works

The most widely discussed recent proposals for the channel networks are *Lightning* and *Raiden*. Both of them are routing payments using the interactive mechanism based on the hashlocked transactions. Lightning is purely a payment network. Raiden, that uses the Ethereum cryptocurrency, can potentially provide state channels. However, this system is still in development, and according to the latest reports[5] the project is "still in the infancy", and the first experimental transactions that were performed concerned the simple payment channels (not the *state* channels). According to other recent reports [25] the developers of this product have focused on constructing a fully functional payment channel under a name *Raiden Minimum Viable Product*, and renamed the full implementation of state channels to "Raiden 2.0" that will be implemented later. State channels have also been recently defined and constructed in [19]. Compared to our work, they do not seem to have an easy interface for handling multiple states in parallel, and for having the two-stage channel update (since this is not needed in [19] for their application).

Payment channels were also constructed in [10]. Other micropayment systems that have been proposed are based on probabilistic payments (see [26, 23, 18, 21]). Our technique for channel virtualization could potentially be used to create such probabilistic payment channels. Channel constructions based on the sequence number maturity (that we use in this paper) have been mentioned already in [22], and recently described in more detail (as "stateful duplex off-chain micropayment channels") in [5]. Payment channels bare some resemblance with the *credit networks* (see, e.g., [14]). It would be interesting to see if our techniques also apply in this area.

A recent idea for micropayments over Bitcoin is based on the "trusted environments" [17]. It would be interesting to see if our protocols can be implemented along the same lines.

2 Our model

We consider *n-party protocols* run by a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of *parties*. We assume that the parties are connected by secure (secret and authenticated) communication channels. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ who can *corrupt* any party P_i, by which we mean that he takes full control over P_i. We provide more details on our concrete choice of the model and assumptions we make in the paragraphs below.

Communication model. We assume a synchronous communication network, where the execution of the protocol happens in rounds. The parties and the adversary are always aware of a given round (in practice one can think of it as equipping them with clocks that are synchronized). Let $\mathsf{time} := \mathbb{N} \cup \{\infty\}$ denote the set of all possible round numbers. We assume that if in round i a party sends a message to another party, then it arrives to it at the beginning of round $i + 1$. At the first sight it may look unrealistic to assume synchronicity in a settings that is inherently asynchronous. We would like to emphasize that this is just a technical assumption that simplifies the description of our protocols. One can think of a "round" as the maximal time that is needed for a message to arrive from one party to another. An assumption that such a bound is known is completely standard in this area (as otherwise no security can be achieved), see, e.g. [2, 4]. We allow the adversary to be *rushing*, i.e., he can decide about the order in which the messages arrive in a given round. For simplicity we assume that computation takes no time and is "atomic".

[5] See `https://goo.gl/e5LgQU`, accessed on May 12, 2017.

Whenever we say that some operation (e.g. delivering a message or simply staying in idle state) *takes time at most* $\tau \in \mathsf{time}$ we mean that it is up to the adversary to decide how long this operation takes (as long as it takes at most τ rounds).

The ledger accounts. We treat the money mechanics in the system in an abstract way, simplifying several things that are not relevant to this paper. In particular we assume that each party P_i has some amount $x_i \in \mathbb{R}_{\geq 0}$ of *coins* in its *account on the ledger*. In our protocols all the operations on parties' accounts are performed only by the contracts (see Sect. 3), and the parties do not directly operate on the accounts. Hence, there is no need to describe how the parties add, remove, or transfer the coins. The way in which the contracts operate on the ledger accounts in described in Sect. 3. In this paper, we ignore the transactions fees.

Public key setup. For simplicity of exposition, we assume that each party P_i is identified by its public key pk_{P_i}, and it knows the corresponding private key sk_{P_i} with respect to some fixed signature scheme $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$. Whenever a party P sends a message m we assume that it also attaches a signature $s := \mathsf{Sign}_{\mathsf{sk}_P}(m)$ to it. In particular the contracts (see Sect. 3) will verify the received signature and only accept m, if s was a valid signature for m. To simplify the description of our protocols and contracts, we will omit to explicitly mention that these signatures are attached to messages sent to the contract, but the reader should keep in mind that the above described process is what happens under the hood.

3 Protocols and contracts

Protocols. Our protocols consist of a number of *subprotocols* (e.g. "closing the channel" on Fig. 5, page 15). The subprotocols can be executed in parallel, subject to some restrictions that will always be explicitly stated (see, e.g., Sect. 4.2.9). Each subprotocol will be defined using *procedures* describing the behavior of the parties and the corresponding *functions* in the *contracts* (we explain the notion of contracts below). The names of the protocols and the procedures will be denoted with a typewriter font.

Every subprotocol involves some number of parties (in this paper this number is always 2 or 3), one of them being the *initiator* of the subprotocol. The name of the subprocedure run by the initiator have a word "`Start`" in it (e.g. `UpdateStart` in Sect. 4.2.6). One can think of this subprocedure as being called by an external user, who also passes to it some parameters as input. When it comes to the other parties participating in the subprotocol, we distinguish two following cases.

1. All the parties that execute a subprotocol agreed beforehand on starting it (with fixed agreed parameters and starting time). In this case we can think of the procedures for all the parties as being called by an external user. Typically procedures for the remaining parties have a word '`Wait`" as a part of their name (e.g. `UpdateWait`).
2. The initiator starts the subprotocol itself without agreeing with the other parties (this can happen, e.g., when a party thinks that the other contract party is behaving maliciously, and wants to close the contract). The other parties will be notified about this via a message from the initiator or from the contract. In this case the procedure for the remaining parties have a word '`React`" as a part of their name (e.g. `CloseReact`). One can think of such procedures as being called internally by the protocol.

If a procedure stops then it returns some value to the calling user. Note that "stopping" a procedure means that just the corresponding thread of the execution stops.

While presenting the subprotocols we will be estimating the time needed for them to execute. We will use the terms *standard* and *pessimistic* execution times. By "standard" time we mean the time needed to execute a given subprotocol if all the contributing parties are honest. "Pessimistic" time concerns the situation when all the parties (except of one) are dishonest and do everything they can to delay the execution. Of course, in reality in most cases the subprotocols will only take the standard time. However, it is also important to know the exact pessimistic times, since strictly speaking, from the honest user's point of view the pessimistic times are the only times that are "guaranteed" by the protocol.

Contracts. We make use of smart contracts, which, informally speaking, are agreements written on the ledger, that can accept coins from the parties, and distribute these coins between the parties, depending on some well-specified conditions. There is no space here to provide an introduction to smart contracts, the reader may consult, e.g., [6, 16]. In this paper, we present contracts in a form of a set of *functions* $\mathsf{fun}_1, \mathsf{fun}_2, \ldots$ written in an informal pseudocode, that operate on contract's storage σ, which is notion that we explain below (their code in Solidity is described in Appx. A).

Strictly speaking the term "contract" will have two different meanings. The first one is the actual contract code (i.e. its algorithmic description), while the second one is the instance of the contract that is running on a ledger (or in a state channel, see Sect. 4.2).[6] Sometimes to avoid confusion we will specifically say "contract code" (in the first case), and "contract instance" (in the second case).

It will be convenient to think of a contract instance as an independent entity that receives coins and messages (in a form of function calls) from the parties and sends coins and messages back to them.[7] Whenever a party P_i "sends y_i coins to a contract C" the amount y_i of coins is removed from P_i's account in the ledger and added to C's account (if P_i does not have enough coins, then it cannot perform this operation). Similarly by saying that a contract "sends y_i coins to a P_i" we mean that y_i coins are added to P_i's account and removed from C's account (this can only happen if a contract owns at least y_i coins).

The reaction time of a contract is at most Δ (where $\Delta \in \mathbb{N}$ is some parameter), i.e. posting a contract on the ledger (see below) can take time at most Δ rounds, functions called by the parties are executed by a contract in time at most Δ, and sending money to C takes time at most Δ. To ease description of our protocols, we assume that the contract's local computation of a contract takes no time and that the messages and coins sent by a contract arrive to the parties immediately. Most of the time the contracts will be in the *idle state*, i.e., they will wait for a function call from a party. A contract can also *terminate*, i.e., it can stop its operations and disappear from the ledger.[8] If the contract terminates then all its unspent money goes to the party that created it.

It is important to emphasize that contracts do not take any actions by themselves, in other words: they need to be triggered by a function call from some party in order to take any action. Between the function calls contract keeps the values of its internal variables in its *storage*[9], which is an attribute tuple (see Sect. 1.2). Hence, the function calls (together with the information about the calling party, time, and the amount of coins) can be viewed as transformations on contract's storage (we elaborate on this in Sect. 3.1). On a high level the life cycle of a contract will look as on Fig. 1

Some function calls can be *ignored*[10] when, e.g., the function inputs are of a wrong form. Whenever some of its functions is called, the contract is informed (by the mechanics of the cryptocurrency) about the identity of the party P that called this function and the actual amount of coins x that

[6] This is similar to a distinction between a *program* and a *process* in the operating systems.

[7] In practice in Solidity, sending messages from the contracts to the parties can be implemented, e.g., using a special construction called "Events" (in our informal description we omit the details of this mechanism).

[8] In Solidity this is performed using the `selfdestruct command.`

[9] For a distinction between "storage" and "memory" see [24], Chapter 6.

[10] In Solidity this translates to executing `throw` instruction.

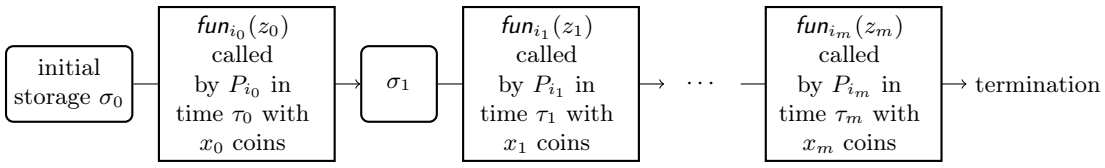


Fig. 1: Life cycle of a contract storage. Above, fun_i's are the contract functions and P_i's are users of the contract.

P attached to this call. Moreover the contract has access to a variable current-time that denotes the exact time when the function is called.[11]

While describing the functions in the contracts we will often say that some function *fun* is "called by party $\hat{P} \in \mathcal{P}$ with $\hat{x} \in \mathbb{R}_{\geq 0}$ coins", where $\hat{P}$ and $\hat{x}$ are read from variables kept in contract's storage (see Fig. 4). This will implicitly mean that the function can be called by *any* P with *any* amount of coins x, but before starting its execution the function checks if $P = \hat{P}$ and $x = \hat{x}$. If some of these checks fail then the function ignores such a call. Otherwise it executes the function code. Analogously, we will sometimes specify time when a function can be called.

Within the contract code each attribute *attr* in contract's storage will be treated as a variable, and referred to as *this*.*attr* (see, e.g., Fig. 4 (b) on page 14). Sometimes we will omit the word "*this*" in and simply write *attr* when it does not lead to a confusion (e.g. we will frequently write "Alice" and "Bob", instead of "*this*.Alice" and "*this*.Bob").

We only consider *bilateral contracts* that are a restricted class of contracts needed for our application. In particular the bilateral contracts are executed by two parties (usually called Alice and Bob) that agreed beforehand on *parameters* π of the contract. Each π is an attribute tuple with attributes id, Alice, Bob, cash (and possibly some other ones), i.e., it is of a form

$$(\pi.\mathsf{id}, \pi.\mathsf{Alice}, \pi.\mathsf{Bob}, \pi.\mathsf{cash}, \ldots),$$

where π.id is a (unique) *identifier* of the contract[12], π.Alice is an identifier of Alice, π.Bob is an identifier of Bob, and π.cash is a function $\pi.\mathsf{cash}\colon \pi.\mathsf{end\text{-}users} \to \mathbb{R}_{\geq 0}$ (where π.end-users is a shorthand for $\{\pi.\mathsf{Alice}, \pi.\mathsf{Bob}\}$). For $P \in \pi.\mathsf{end\text{-}users}$ the value $\pi.\mathsf{cash}(P)$ is called *the amount of cash of* P *in the contract*. Moreover we define $\pi.\mathsf{other\text{-}party}\colon \mathcal{P} \to \mathcal{P}$ as $\pi.\mathsf{other\text{-}party}(\pi.\mathsf{Alice}) := \pi.\mathsf{Bob}$ and $\pi.\mathsf{other\text{-}party}(\gamma.\mathsf{Bob}) := \pi.\mathsf{Alice}$.

3.1 Functions as storage transformations

As we already mentioned, one can view function calls in a contract code as transformations of contract's storage. More precisely, let Init be a function that on input (τ, π) (where τ denotes time when the contract is created, and π are the initial parameters) outputs the contents $\sigma := \mathsf{Init}(\tau, \pi)$ of the storage of the contract constructed as follows: first let $\sigma := \pi$, and then set $\sigma.\mathsf{init\text{-}time} := \tau$.[13]

Further transformations of storage of a contract C are captured by a function Eval defined in the following way. Let σ be contract's storage, let *fun* be one of the functions in a contract code C and let $P \in \sigma.\mathsf{end\text{-}users}$. Moreover let $\tau \in \mathbb{N}$, $z \in \{0,1\}^*$, and $x \in \mathbb{R}_{\geq 0}$. Then the value of $\mathsf{Eval}^{\mathsf{C}}(\sigma; P, x, \tau; \mathit{fun}, z)$ is equal to

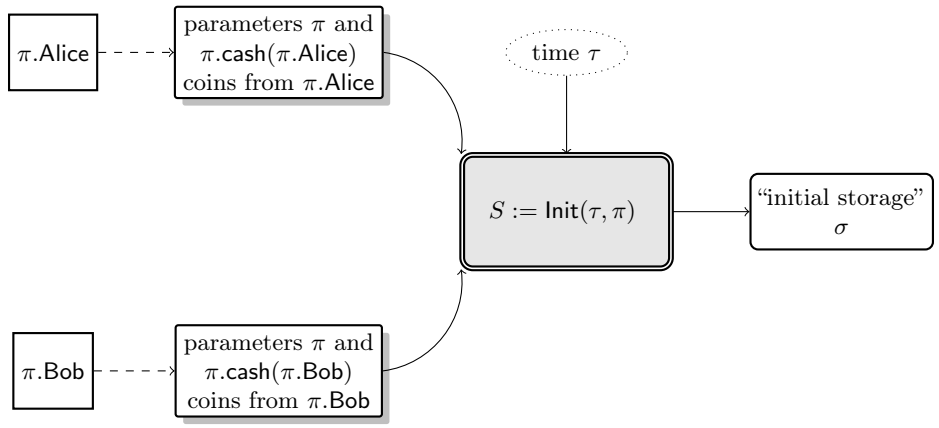
$$(\sigma'; m_a, y_a; m_b, y_b) := \mathsf{Eval}^{\mathsf{C}}(\sigma; P, x, \tau; \mathit{fun}, z), \tag{1}$$

[11] In Solidity information about P, current-time, and x is passed to a contract via special fields denoted `msg.sender`, `block.timestamp`, and `msg.value`, respectively (see `http://solidity.readthedocs.io/en/develop/units-and-global-variables.html`).

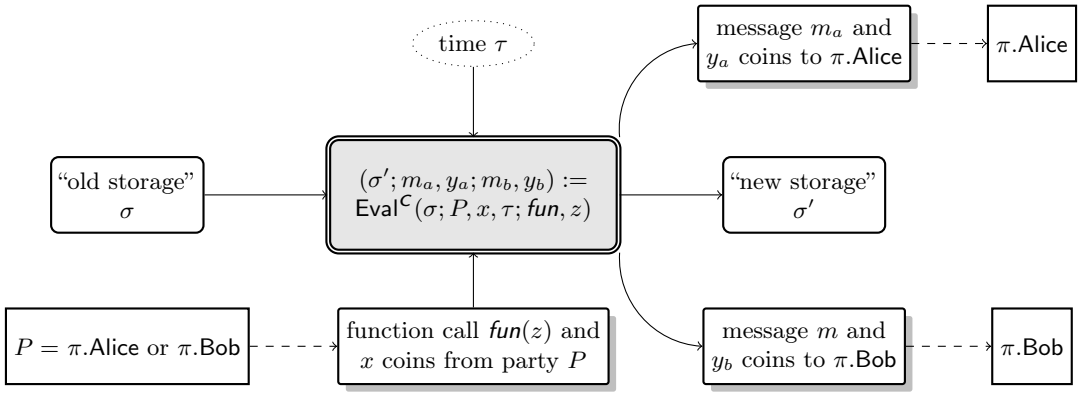
[12] In Solidity this is called an "address" of a contract.

[13] Such an initialization will also have an effect that $\pi.\mathsf{cash}(\pi.\mathsf{Alice})$ and $\pi.\mathsf{cash}(\pi.\mathsf{Bob})$ coins are removed from π.Alice's and π.Bob's deposits in the ledger or in a state channel (see Sect. 4.2), respectively.

if a function *fun* called by P in time τ with parameter z and x coins when C's storage is σ, results in new storage σ', and messages $m_a \in \{0,1\}^*$ (with y_a coins) and $m_b \in \{0,1\}^*$ (with y_b coins) sent to the parties that are running the contract ($\sigma.\mathsf{Alice}$ and $\sigma.\mathsf{Bob}$, respectively). If the contract has terminated then we assume that $\sigma' = \mathsf{terminated}$ in (1). The diagrams presenting computation of Init and Eval functions are depicted on Fig. 2.



(a)



(b)

Fig. 2: Diagrams presenting computation of (a) Init and (b) Eval functions.

3.2 Creating contracts on the ledger

In this paper we consider two types of contracts. The first ones are standard contracts that are posted on the ledger, which are sometimes called *ledger contracts*. The other type of contracts are the *nanocontracts* that will be executed within the state channels (see Sect. 4.2). The ledger contracts will have a special form allowing them to be easily initialized on the ledger, which we explain below.

The bilateral ledger contract creation procedure is always initiated by $\pi.\mathsf{Alice}$ who passes to a contract code C parameters $\pi \in \{0,1\}^*$ and posts contract instance of C on the public ledger

(the contract appears on the ledger in time at most Δ). Technically "informing" C about the value π is done by calling a special function called *contract's constructor* that is a part of the contract description and is denoted by C (i.e. it has the same name as C, but is written in the non-italic font). The constructor is called by party $P = \pi.\mathsf{Alice}$ on input π together with $x_a = \pi.\mathsf{cash}(\pi.\mathsf{Alice})$ coins. If P is malicious then it can happen that $P \neq \pi.\mathsf{Alice}$ or $x_a \neq \pi.\mathsf{cash}(\pi.\mathsf{Alice})$, in which case the contract ignores this message. Otherwise the contract appears on the ledger, and from now on it is denoted C_{id}, where $id = \pi.\mathsf{id}$. The fact that C_{id} appeared on the ledger is communicated to the parties by a message $\mathsf{initializing}(\pi)$ (sent by this contract). In a reaction to this, $\pi.\mathsf{Bob}$ immediately calls a function $\mathsf{confirm}()$ in C_{id} together with $x_b = \pi.\mathsf{cash}(\pi.\mathsf{Bob})$ coins. Once the C_{id} receives this function call from $\pi.\mathsf{Bob}$, it checks if x_b indeed is equal to $\pi.\mathsf{cash}(\pi.\mathsf{Bob})$, and if yes, it sends a message $\mathsf{initialized}$ to $\pi.\mathsf{Alice}$ and $\pi.\mathsf{Bob}$ and goes idle.

If $\pi.\mathsf{Alice}$ does not receive the $\mathsf{initialized}$ message from C_{id} within time Δ from the time when she received the $\mathsf{initializing}$ message (which happens if $\pi.\mathsf{Bob}$ did not call the $\mathsf{init}(\pi)$ function message together with x_b coins), then she needs to ask C_{id} to refund her money. This is done by sending a message $\mathsf{refund}()$ to C_{id}. The contract replies by sending back to $\pi.\mathsf{Alice}$ her x_a coins together with a message $\mathsf{refunded}$ and terminates. The details of this procedure are presented on Figures 4 and 3.

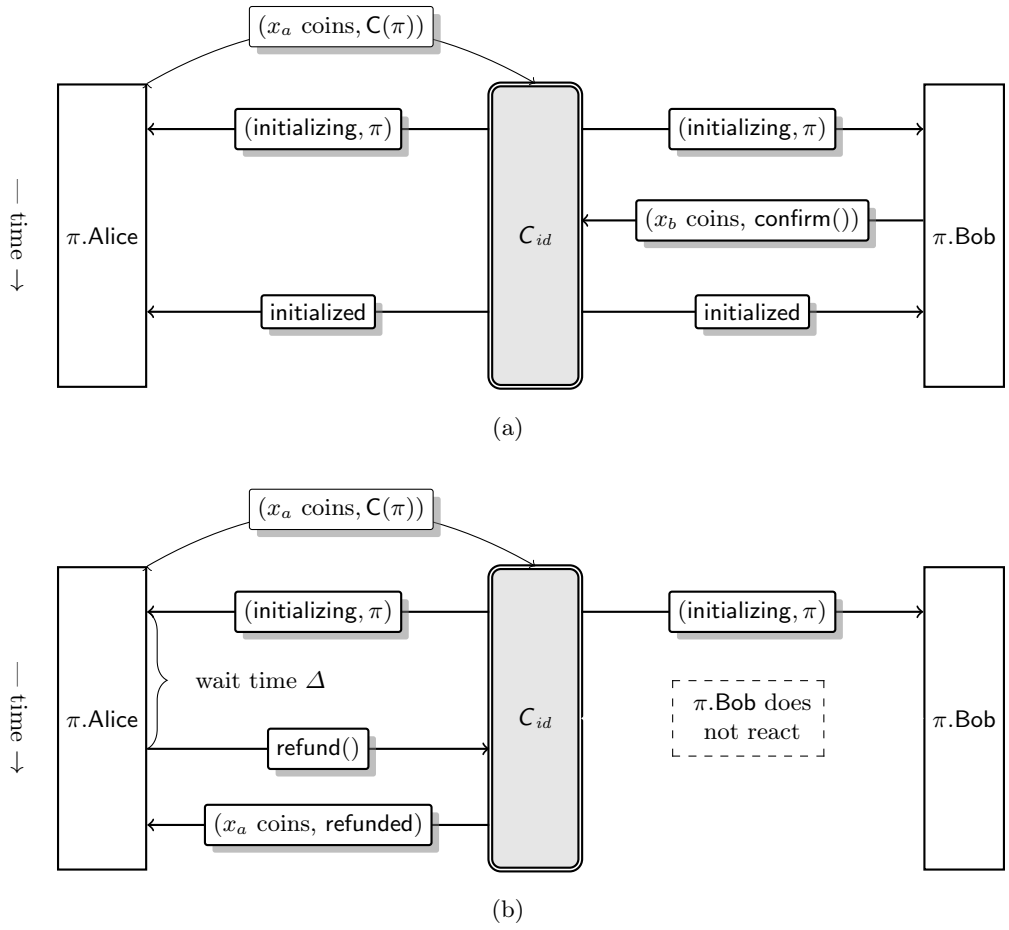


Fig. 3: The message flow between the subprocedure and the contract of Fig. 4: (a) the message flow in a successful execution of the scheme, and (b) the message flow in an *unsuccessful* execution of the scheme in case $\pi.\mathsf{Bob}$ did not call the $\mathsf{confirm}$ function.

4 The Channels

In this section we describe the channel constructions. We start with describing simple payment channels (in Sect. 4.1), which we mainly do for making the reader familiar with our framework and the formalism that will be used to describe our contributions. Then, in Sect. 4.2 we describe the multistate channels (our extension of the state channels). In Section 4.3 we give a full description of the protocols and contracts that are needed for virtual channels, which is the main contribution of this paper.

4.1 Payment channels

Recall (see Sect. 1) that payment channels enable its users to transfer coins between each other without the need of touching the blockchain. In other words, payment channels are protocols that mainly operate off-chain, but can be realized on the blockchain by its users in case of dispute. The construction that we present below is based on the idea of "sequence numbers" (see [22]), which, in our description, are referred to as "version numbers".

4.1.1 Channel syntax. A *basic payment channel over a set of parties* $\mathcal{P}$ is an attribute tuple γ of the form

$$\gamma = (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}), \tag{2}$$

where $\gamma.\mathsf{id} \in \{0,1\}^*$ is called the *identifier of* γ and values $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ are two distinct elements of $\mathcal{P}$ called the *end-users of* γ (we will also call them simply Alice and Bob, when γ is clear from the context). We will use shorthands: $\gamma.\mathsf{end\text{-}users}$ and $\gamma.\mathsf{other\text{-}party}$ defined in the same way as we did it for contract parameters (see Sect. 3). The attribute $\gamma.\mathsf{cash}$ is a function $\gamma.\mathsf{cash} : \gamma.\mathsf{end\text{-}users} \to \mathbb{R}_{\geq 0}$ and for $P \in \gamma.\mathsf{end\text{-}users}$ the value $\gamma.\mathsf{cash}(P)$ is called *the amount of cash of* P *in* γ. *Adding* x *coins to* P*'s deposit in* γ is defined as setting $\gamma.\mathsf{cash}(P) := \gamma.\mathsf{cash}(P) + x$ (*removing* x *coins* is defined analogously). We will also say that γ *is established between the parties in* $\gamma.\mathsf{end\text{-}users}$.

We assume that every correctly formatted $\gamma.\mathsf{id}$ contains information about the identities of $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ plus two nonces. More precisely: every $\gamma.\mathsf{id}$ is of a form: $\gamma.\mathsf{id} = (\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, nonce_1, nonce_2)$, where $nonce_i$'s are some long (160 bits, say) random strings, such that $nonce_1$ is chosen when a new instance of the system is started (and it is fixed in every $\gamma.\mathsf{id}$ in this instance), and $nonce_2$ is chosen freshly for every new channel. Including $\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}$, and $nonce_2$ is needed to guarantee that the identifiers of the channels are unique, i.e., no two channels have the same identifier (if at least one of their end-users is honest), while $nonce_1$ is needed to guarantee that $\gamma.\mathsf{id}$ is unique on the blockchain. This is important to ensure because the contract that we create for handling γ will have the same identifier as γ (see below), and hence $\gamma.\mathsf{id}$ has to be "globally" unique. To simplify the description, in the sequel we will not mention this requirement on the format of $\gamma.\mathsf{id}$ explicitly.

The basic payment channels are implemented by a protocol `PaymentChannels`. Each channel γ will have its corresponding contract $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ on the ledger that will interact with both $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$. A channel is *active* if the corresponding $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ exists on the ledger and it has not been terminated. In our protocols, each party $P \in \mathcal{P}$ maintains a channel space Γ^P that contains all the active basic channels that this party established. For every id the values of attributes $\Gamma^P(id).\mathsf{id}, \Gamma^P(id).\mathsf{Alice}$, and $\Gamma^P(id).\mathsf{Bob}$ will never change. The only value that can change is the attribute $\Gamma^P(id).\mathsf{cash}$ (except that the sum of $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ and $\gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ will changes). When it is more convenient we will simply write (x_a, x_b) to represent $\gamma.\mathsf{cash}$ (where $(x_a, x_b) := (\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}))$. The current value of $\gamma.\mathsf{cash}$ will also be called *channel's*

Protocol for initializing contract $\mathcal{C}$

To initialize $\mathcal{C}$ with parameter π the parties execute the following protocol (let $id := \pi.\mathsf{id}$).

Procedure $\texttt{InitializeStart}(\mathcal{C}; \pi)$ run by $\pi.\mathsf{Alice}$

1. Initialize $\mathcal{C}_{id}$ using its constructor C with parameter π and post it on the ledger with $x_a = \pi.\mathsf{cash}(\pi.\mathsf{Alice})$ coins.
2. Upon receiving a message $(\mathsf{initializing}, \pi)$ from $\mathcal{C}_\pi$ wait time at most Δ to receive a message $\mathsf{initialized}$ from this contract.

Procedure $\texttt{InitializeWait}(\mathcal{C}; \pi)$ run by $\pi.\mathsf{Bob}$

3. Upon receiving a message $(\mathsf{initializing}, \pi)$ from $\mathcal{C}_\pi$ reply by calling a function $\mathsf{confirm}()$ together with $x_b = \pi.\mathsf{cash}(\pi.\mathsf{Bob})$ coins.
4. Upon receiving a message $\mathsf{initialized}$ from $\mathcal{C}_{id}$ conclude that the channel has been created, i.e., return $\mathsf{initialized}$.

Back to procedure $\texttt{InitializeStart}(\mathcal{C}; \pi)$ run by $\pi.\mathsf{Alice}$

5. If you receive a message $\mathsf{initialized}$ from $\mathcal{C}_{id}$ within time Δ conclude that the channel has been created, i.e. return $\mathsf{initialized}$.
6. Otherwise call a function $\mathsf{refund}()$ in $\mathcal{C}_{id}$ and receive back a message $\mathsf{refunded}$ together with x_a coins. Return $\mathsf{not\text{-}initialized}$.

(a)

$\mathcal{C}_{id}$

Constructor $\mathsf{C}(\pi)$ called by $P \in \mathcal{P}$ with $x > 0$ coins

If $P \neq \pi.\mathsf{Alice}$ or $x \neq \pi.\mathsf{cash}(\pi.\mathsf{Alice})$ then ignore this message. Otherwise let your storage be equal to $\mathsf{Init}(\mathsf{current\text{-}time}, \pi)$ and additionally set $\mathit{this}.\mathsf{initialization} := \mathsf{true}$.
Send a message $(\mathsf{initializing}, \pi)$ to Alice and Bob and go idle.

Function $\mathsf{confirm}()$ called by Bob with $\mathsf{cash}(\mathsf{Bob})$ coins

1. If $\mathsf{current\text{-}time} > \mathsf{init\text{-}time} + \Delta$ or flag $\mathit{this}.\mathsf{initialization}$ is undefined then ignore this call.
2. Otherwise send a message $\mathsf{initialized}$ to Alice and Bob, remove variable $\mathit{this}.\mathsf{initialization}$ from your storage and go idle.

Function $\mathsf{refund}()$ called by Alice

1. If $\mathsf{current\text{-}time} \leq \mathsf{init\text{-}time} + \Delta$ or flag $\mathit{this}.\mathsf{initialization}$ is undefined then ignore this call.
2. Otherwise send a message $\mathsf{refunded}$ to Alice and terminate the contract.

$\cdots$ other functions in the contract (they can be called only if flag $\mathit{this}.\mathsf{initialization}$ is undefined) $\cdots$

(b)

Fig. 4: The "Channel creation" subprocedure: (a) the subprotocol for the parties, and (b) the corresponding subcontract of $\mathcal{C}_{id}$. Strictly speaking, $\mathsf{Alice}, \mathsf{Bob}$ and cash are variables from contract's storage, and should be referred to as $\mathit{this}.\mathsf{Alice}$, $\mathit{this}.\mathsf{Bob}$ and $\mathit{this}.\mathsf{cash}$. We omit the "*this*" keyword, following the convention described in Sect. 3.

.

state. The *initial form of a channel with identifier id* is equal to γ with all attributes defined as when the channel was created.

4.1.2 Overview of the scheme. The protocol `PaymentChannels` consists of subprotocols called "creating a channel", "updating the channel", and "closing the channel", described in Sections 4.1.3, 4.1.4, and 4.1.5, respectively. The protocol uses a contract *PaymentContract* consisting of functions for creating a ledger contract (the contract constructor, confirm, refund), and functions close and finalize, used for closing the contract (the code of these functions is described later, see Sect. 4.1.5).

The general structure of the scheme, including the names of the procedures and the message types that are sent between the parties and the contract, is depicted in Fig. 5. The protocol uses a contract *PaymentContract*. The list of functions in this contract in presented on Fig. 6.

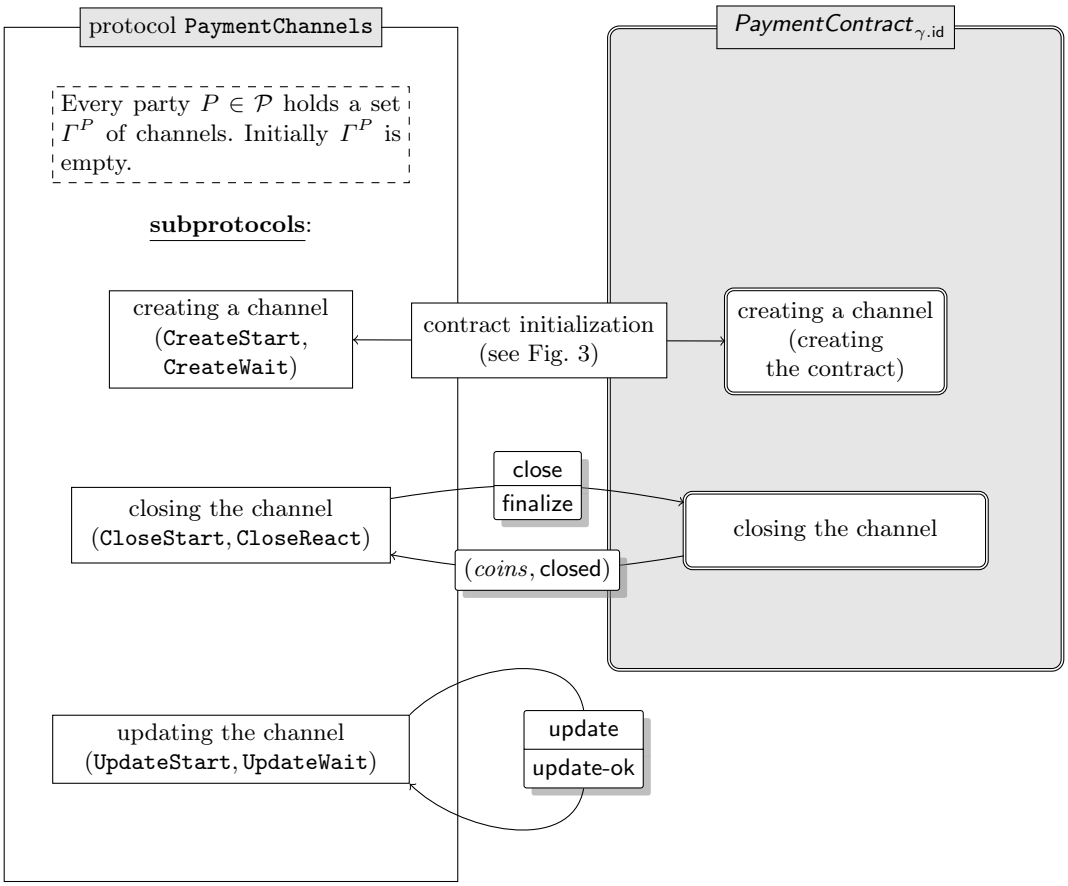


Fig. 5: The general structure of the scheme for basic payment channels, including the messages that are sent between the parties and the contract, and the names of the procedures.

Internally, the protocol extends the channel notation described in Sect. 4.1.1 by adding to γ attributes $\gamma.\mathsf{ver\text{-}num} \in \mathbb{N}$ and $\gamma.\mathsf{sign} \in \{0,1\}^*$ (invisible from the point of view of the external user[14]). Hence γ has now the following form (cf. (2)):

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

[14] One can think of $\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}$, and $\gamma.\mathsf{cash}$ as "public variables", and of $\gamma.\mathsf{ver\text{-}num}$ and of $\gamma.\mathsf{sign}$ as "private variables".

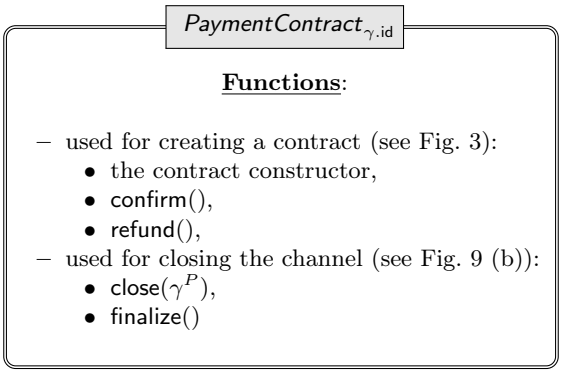


Fig. 6: Functions in the *PaymentContract*.

The role of these variables will be explained in more detail later. For a moment let us just say that $\gamma.\mathsf{ver\text{-}num}$ corresponds to the "version number" of the channel (it will be incremented with each update of channel's state), and $\gamma.\mathsf{sign}$ will be a signature of $\gamma.\mathsf{other\text{-}party}(P)$ on $(\gamma.\mathsf{id}, \gamma.\mathsf{cash}, \gamma.\mathsf{ver\text{-}num})$. A tuple

$$(\gamma.\mathsf{id}, \gamma.\mathsf{cash}, \gamma.\mathsf{ver\text{-}num}, \gamma.\mathsf{sign})$$

is be called *a version of γ's state signed by $P \in \gamma.\mathsf{end\text{-}users}$* if $\gamma.\mathsf{sign}$ is a signature of P on $(\gamma.\mathsf{id}, \gamma.\mathsf{cash}, \gamma.\mathsf{ver\text{-}num})$. If $\gamma.\mathsf{ver\text{-}num}$ is equal to 0 then we do not require $\gamma.\mathsf{sign}$ to do defined, but, for the sake of consistency, also in this case we say that $(\gamma.\mathsf{id}, \gamma.\mathsf{cash}, \gamma.\mathsf{ver\text{-}num})$ is "a version of γ's state signed by any $P \in \gamma.\mathsf{end\text{-}users}$".

4.1.3 Creating a channel. At any point in time parties P_a and P_b can decide to create a channel γ such that $\gamma.\mathsf{Alice} = P_a$ and $\gamma.\mathsf{Bob} = P_b$. This can be done only if no channel with identifier $\gamma.\mathsf{id}$ has been created before, and if $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ have $x_a := \gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ and $x_b := \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ unspent coins on the ledger (respectively). It is the parties' responsibility to choose γ such that $\gamma.\mathsf{id}$ has not been used before. Recall that we assumed that $\gamma.\mathsf{id}$ contains information about the identities of the parties that created it. Hence, as long as at least one of them is honest, there is no risk that another correctly formatted channel with the same identifier has been already created. We assume that the parties agree beforehand on all the attributes of γ and on the exact round τ when the creation procedure starts. To start the channel $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ run the protocol for initializing *PaymentContract* with parameter $\pi = \gamma$ (see Fig. 4 on page 14).

4.1.4 Updating the channel. After the channel has been created, Alice and Bob can change its *state*, i.e., the value of the pair $(\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}))$, as long as the sum of $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ does not change. This is done via a procedure called "updating the channel" and does *not* require communicating with *PaymentContract*$_{\gamma.\mathsf{id}}$ (or the ledger). This mechanism is used for performing "offline" payments between the parties. For instance, if the current channel state is (x_a, x_b) and Alice wants to transfer 1 coin to Bob, the parties create a new state of the channel that has state $(\tilde{x}_a, \tilde{x}_b) = (x_a - 1, x_b + 1)$. The state of γ can be updated multiple times. The mechanism behind the channel update procedure and the *PaymentContract*$_{\gamma.\mathsf{id}}$ guarantees that if at some point one of the parties wants to close out the channel, and receive back the funds blocked in the channel, then it is always the latest state that will count.

At a technical level, this is implemented in the following way (a more detailed description, including the message flow, is depicted on Fig. 8). Consider some fixed channel γ that has not been

Subprotocol for creating a channel

Procedure `CreateStart`(γ) run by γ.Alice

1. Add an attribute ver-num to γ, and let γ.ver-num $= 0$.
2. Let w_a := `InitializeStart`(*PaymentContract*; γ).
3. If w_a = initialized then store γ in Γ and output created.
4. Otherwise (if w_a = not-initialized) output not-created.

Procedure `CreateWait`(γ) run by γ.Bob

1. Add an attribute ver-num to γ, and let γ.ver-num $= 0$.
2. Let w_b := `InitializeWait`(*PaymentContract*; γ).
3. If w_b = initialized then store γ in Γ and output created.
4. Otherwise (if w_b = not-initialized) output not-created.

Fig. 7: The "Channel creation" subprocedure. Recall that `InitializeStart` and `InitializeWait` are procedures for contract creation, described on Fig. 4. We let $w_a, w_b \in \{\text{initialized}, \text{not-initialized}\}$ be the outputs of these procedures.

updated yet. Let (x_a, x_b) be its current state, and let id be its identifier. Suppose that Alice and Bob agreed (via some other communication channel) that they both want to change the state of γ to $(\tilde{x}_a, \tilde{x}_b)$ (where $\tilde{x}_a + \tilde{x}_b = x_a + x_b$), and that one of them, denoted P will be the *initiator* of the update subprotocol. First P constructs γ^P as equal to γ on attributes id and users, and

- $(\gamma^P.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma^P.\mathsf{cash}(\gamma.\mathsf{Bob})) := (\tilde{x}_a, \tilde{x}_b)$,
- $\gamma^P.\mathsf{ver\text{-}num} := 1$, and
- $\gamma^P.\mathsf{sign}$ equal to P's signature on $(\gamma^P.\mathsf{id}, \gamma^P.\mathsf{cash}, 1)$.

Then P sends a message $(\mathsf{update}, \gamma^P)$ to P'. If γ^P is correctly signed by P, then P' replaces γ in $\Gamma^{P'}$ with γ^P and sends back to P a message $(\mathsf{update\text{-}ok}, \gamma^{P'})$ where $\gamma^{P'}$ is equal to γ^P on all attributes, except that $\gamma^{P'}.\mathsf{sign}$ is equal to P''s signature on $(\gamma^{P'}.\mathsf{id}, \gamma^{P'}.\mathsf{cash}, \gamma^{P'}.\mathsf{ver\text{-}num})$.

Upon receiving this message, party P checks if $\gamma^{P'}$ is correctly signed by P', and if yes then she replaces γ in Γ^P with γ^P. Note that after these steps are performed, the only difference between $\Gamma^P(id)$ and $\Gamma^{P'}(id)$ is on the sign attribute. As a result of this procedure, each party has a signature of the other party on the updated state $(\tilde{x}_a, \tilde{x}_b)$ of the channel, and when the channel is closed (see Sect. 4.1.5) she can present it to the $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ as an evidence that the other party agreed for the update.

Further updates are performed in the same way, except that the value of the parameter ver-num is incremented with each update. More precisely, let γ be the latest value of $\Gamma^P(id)$ and suppose P and P' agreed to update the channel to a new state $(\tilde{x}_a, \tilde{x}_b)$. Then everything is done exactly as before, except that $\gamma^P.\mathsf{ver\text{-}num}$ gets incremented by 1. The idea behind the increasing "γ.ver-num" numbers is pretty straightforward: when the channel is closed the signed state with the higher ver-num will count as the valid one. Therefore a malicious party cannot trick $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ to accept some older version of the state (as the other party will always have a chance to send to the contract the state with the higher γ.ver-num). We describe it in more detail in Sect. 4.1.5.

In the description above we assumed that the parties will always send the correct messages when they are supposed to do so. For example, we did not describe what happens when the signatures are incorrect, or when P' does not reply to P with the update-ok message. Note that an honest P' always replies with the update-ok message, since we assumed the parties agreed beforehand that the protocol will be initiated by P. Such cases will be handled in the following way. A party (P or P') that notices such an incorrect behavior of the other party will simply conclude that this party is corrupt. Usually after such a conclusion a party will simply proceed to the "channel closing procedure" (see Sect. 4.1.5 below).

Subprotocol for updating the channel

Let id be a channel identifier of some active channel. To update the channel with identifier id to a new state $(\tilde{x}_a, \tilde{x}_b)$ its end-users P and P' proceed as follows:

Procedure $\texttt{UpdateStart}(id, \tilde{x}_a, \tilde{x}_b)$ run by P:

1. Let γ^P be equal to $\Gamma^P(id)$ on attributes id and users, let $\gamma^P.\mathsf{cash}$ be defined as $(\gamma^P.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma^P.\mathsf{cash}(\gamma.\mathsf{Bob})) := (\tilde{x}_a, \tilde{x}_b)$, and let $\gamma^P.\mathsf{ver\text{-}num} = \Gamma(id).\mathsf{ver\text{-}num} + 1$. Moreover let $\gamma^P.\mathsf{sign} := \mathsf{Sign}_{\mathsf{sk}_P}(\gamma^P.\mathsf{id}, \gamma^P.\mathsf{cash}, \gamma^P.\mathsf{ver\text{-}num})$.
2. Send a message $(\mathsf{update}, \gamma^P)$ to P'.
3. Wait for 1 round.

Procedure $\texttt{UpdateWait}(id, \tilde{x}_a, \tilde{x}_b)$ run by P':

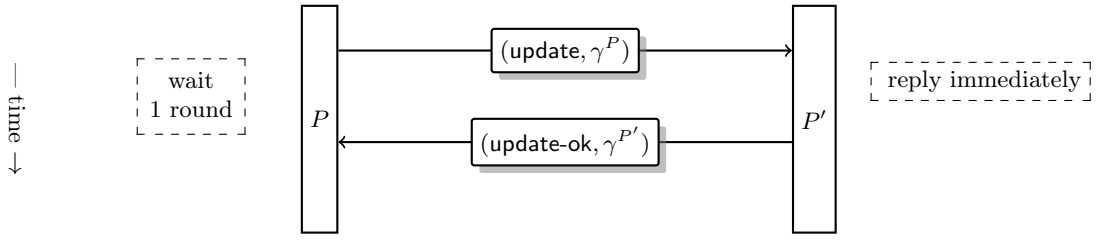
4. Upon receiving message $(\mathsf{update}, \gamma^P)$ check if $(\gamma^P.\mathsf{id}, \gamma^P.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma^P.\mathsf{cash}(\gamma.\mathsf{Bob})) = (id, \tilde{x}_a, \tilde{x}_b)$, if $\gamma^P.\mathsf{ver\text{-}num} = \Gamma^{P'}(id).\mathsf{ver\text{-}num} + 1$, and if γ^P is correctly signed by P. If yes, then:
   - let $\Gamma^{P'}(id) := \gamma^P$,
   - let $(\gamma^{P'}.\mathsf{id}, \gamma^{P'}.\mathsf{cash}, \gamma^{P'}.\mathsf{ver\text{-}num}) := (id, \tilde{x}_a, \tilde{x}_b, \gamma^P.\mathsf{ver\text{-}num})$,
   - let $\gamma^{P'}.\mathsf{sign} := \mathsf{Sign}_{\mathsf{sk}_{P'}}(\gamma^P.\mathsf{id}, \gamma^P.\mathsf{cash}, \gamma^P.\mathsf{ver\text{-}num})$,
   - send a message $(\mathsf{update\text{-}ok}, \gamma^{P'})$ to P' and return $\mathsf{channel\text{-}updated}$.

   Otherwise return $\mathsf{channel\text{-}not\text{-}updated}$.

Back to procedure $\texttt{UpdateStart}(id, \tilde{x}_a, \tilde{x}_b)$ run by P:

5. If you do not receive a message $(\mathsf{update\text{-}ok}, id, \gamma^{P'})$ from P' then return $\mathsf{channel\text{-}not\text{-}updated}$.
6. Otherwise (if you receive such a message) check if $\gamma^{P'}$ and γ^P are equal on attributes id and cash, and if $\gamma^{P'}$ is correctly signed by P'. If yes, then let $\Gamma^{P'}(id) := \gamma^P$ and return $\mathsf{channel\text{-}updated}$.
   Otherwise return $\mathsf{channel\text{-}not\text{-}updated}$.

(a)



(b)

Fig. 8: The "Channel update" subprocedure: (a) the subprotocol for the parties, (b) a message flow in a successful execution of the subprocedure.

4.1.5 Closing the basic payment channel. When a party $P \in \gamma.\mathsf{end\text{-}users}$ wants to close the channel with identifier id, she needs to "let the contract $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ know" about the latest state of channel γ. This is done via a procedure that we call "state registration". Let $\gamma^P := \Gamma^P(id)$. Party P calls a function $\mathsf{close}(\gamma^P)$ in $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$. The contract checks if γ^P is correctly signed by $P' := \gamma.\mathsf{other\text{-}party}(P)$ and if the sum of parties' coins in γ^P is equal to the sum S of their coins in the original channel γ (that the contract received when the channel was established). If it is not, then the contracts ignores this call. Otherwise it sends a message $\mathsf{closing}$ to both $\gamma.\mathsf{end\text{-}users}$.

Party P' replies immediately by calling a function $\mathsf{close}(\gamma^{P'})$, where $\gamma^{P'} := \Gamma^{P'}(id)$. If within time Δ the contract receives this call from P' then it checks if if the sum of parties' coins in γ^P is equal to the S and if $\gamma^{P'}$ is is correctly signed by P. If yes, then it lets γ^* be equal to the one of γ^P and $\gamma^{P'}$ that has the higher version number (ties are resolved in favor of P)[15], more precisely it lets:

$$\gamma^* := \begin{cases} \gamma^P \text{ if } \gamma^P.\mathsf{ver\text{-}num} \geq \gamma^{P'}.\mathsf{ver\text{-}num} \\ \gamma^{P'} \text{ otherwise} \end{cases}$$

If party P' does not make such a call within time Δ then only the parameters provided by P count (and hence $\gamma^* = \gamma^P$). Informally, this means that P' had a chance to present his version of the state, but failed to do so within time Δ. Since the contract never does anything "by itself", P has to wake up $\mathit{PaymentContract}_{\gamma}$ after this time passed by calling a function "$\mathsf{finalize}()$". The money is then distributed according to $\gamma^*.\mathsf{cash}$, i.e., each P receives $\gamma^*.\mathsf{cash}(P)$ coins. The whole procedure is depicted on Fig. 9. The corresponding message flows are depicted on Fig. 10.

4.2 Multistate channels

The multistate channels extend the payment channels by allowing their users to execute "inside of them" a bilateral contract code C (see Sect. 3). More concretely, a multistate channel, in addition to Alice's and Bob's cash, contains some number of attribute tuples called *nanocontracts*, each of them being of a form:

$$\nu = (\nu.\mathsf{nid}, \nu.\mathsf{blocked}, \nu.\mathsf{storage}), \tag{3}$$

where $\nu.\mathsf{nid} \in \{0,1\}^*$ is the *identifier of this nanocontract* and $\nu.\mathsf{blocked}$ is a function referring to the amount of money *blocked* in the nanocontract by the parties $\gamma.\mathsf{end\text{-}users}$. More formally, it has a type: $\nu.\mathsf{blocked}\colon \gamma.\mathsf{end\text{-}users} \to \mathbb{R}$, where for $P \in \gamma.\mathsf{end\text{-}users}$ the value $\nu.\mathsf{blocked}(P)$ is called the amount of *coins blocked in ν by P*. The *total* amount of *coins blocked in ν* is equal to $\nu.\mathsf{blocked}(\gamma.\mathsf{Alice}) + \nu.\mathsf{blocked}(\gamma.\mathsf{Bob})$, and is denoted $\nu.\mathsf{total\text{-}blocked}$. Finally $\nu.\mathsf{storage}$ is an attribute tuple that contains the *storage* of nanocontract ν. This value will be interpreted by the bilateral contract C using the $\mathsf{Eval}^{\mathsf{C}}$ function in the way described in Sect. 3.1.

We will provide more details on the semantics of the nanocontracts in a moment, for now let us just focus on the syntax of multistate channels. Technically, the nanocontracts in a channel γ will be stored in a set nspace called *nanocontracts space*. This set will have a property that for every *nid* there exists at most one element $\nu \in \mathsf{nspace}$ with $\nu.\mathsf{nid} = nid$. Hence nspace will also be viewed as a partial function that assigns to every such *nid* the value $\mathsf{nspace}(nid)$ (equal to such a ν) or $\perp$ (if such a ν does not exist).

Formally a *basic multistate channel over a set of players $\mathcal{P}$* is an attribute tuple γ of a form

$$\gamma = (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{nspace}), \tag{4}$$

where $\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}$ and $\gamma.\mathsf{cash}$ are as in the payment channels (see Sect. 4.1) and $\gamma.\mathsf{nspace}$ is a nanocontract space. We assume that no two nanocontracts in the system will have the same

[15] This does not matter, since a tie will never happen if at least one party in $\gamma.\mathsf{end\text{-}users}$ is honest.

identifier. E.g. for every $\nu \in \gamma.\mathsf{nspace}$ we have that $\nu.\mathsf{nid}$ contains information about the identifier $\gamma.\mathsf{id}$ (e.g. it is of a form $\nu.\mathsf{nid} = (\gamma.\mathsf{id}, nid')$ (where nid' is some string). The *value of* γ is is equal to

$$\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}) + \sum_{\nu \in \gamma.\mathsf{nspace}} \nu(nid).\mathsf{total\text{-}blocked},$$

where the sum is taken over all nid's such that $\gamma.\mathsf{nspace}(nid) \neq \bot$.

A *basic multistate channel space* is defined in the same way as the the basic payment channel space (see Sect. 4.1.1), and is also denoted with Γ. When a nanocontract is created, $x_a := \nu.\mathsf{blocked}(\gamma.\mathsf{Alice})$ and $x_b := \nu.\mathsf{blocked}(\gamma.\mathsf{Bob})$ coins are removed from the deposits of $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ (respectively) in γ. The value $\nu.\mathsf{storage}$ will be set to $\mathsf{Init}(\tau, \pi)$ (where τ is the current time, and π denotes the parameters of the contract). We will refer to the current value of ν as *nanocontract's state*. A user $P \in \gamma.\mathsf{end\text{-}users}$ can change the state of a nanocontract by "executing" a call to a function fun from C (with a parameter z) in time τ. This will be done by evaluating function $\mathsf{Eval}^{\mathsf{C}}$ (cf. (1)) in a way that is similar to the execution of a contract C over the ledger. The main difference is that the financial operations are not performed on the users' deposits on the ledger, but on parties' deposits in γ. More precisely, if $(\sigma'; m_a, y_a; m_b, y_b)$ is the output of $\mathsf{Eval}^{\mathsf{C}}$ then σ' will become the new storage of the nanocontract, Alice and Bob will receive messages m_a and m_b (respectively), their respective amounts of coins blocked in ν will be decreased by y_a and y_b, and "moved" to their respective deposits in γ (for technical details see Sect. 4.2.5). Before the execution starts the ledger needs to determine what is the current state of a nanocontract. This is done in a procedure called "nanocontract registration" (see Sect. 4.2.4).

Typically, such an execution will happen only if the parties disagree. As long as the parties are not in a dispute, they can freely update nanocontract's state. This means, in particular, that they may transfer additional coins from $\gamma.\mathsf{cash}$ to the nanocontract, move coins between different nanocontracts, or change its state. Hence, they can also agree to "execute a function in a peaceful way", i.e., each of them can compute $\mathsf{Eval}^{\mathsf{C}}$ locally and update the nanocontract according to the output of this computation. We emphasize that in case of such a "peaceful" execution the nanocontract will not go on the ledger, but the execution will merely happen locally by the parties and technically realized by updating the state of the nanocontract. The procedure of first trying to execute a nanocontract peacefully, and then (if it fails) running the standard execution is called "optimistic execution" (it is described in Sect. 4.2.8).

The values $x_a := \nu.\mathsf{blocked}(\gamma.\mathsf{Alice})$ and $x_b := \nu.\mathsf{blocked}(\gamma.\mathsf{Bob})$ can change during the lifetime of the nanocontract (as a result of nanocontract updating, or nanocontract function execution). We will allow x_a and x_b to be negative numbers, as long as their sum is non-negative. This is interpreted as follows. A *positive* value x_a denotes the fact that Alice "invested" x_a coins into the nanocontract. In other words: x_a of her coins (that she has in channel γ) are blocked in the nanocontract, and if the nanocontract is closed with such a value of x_a, then Alice *loses* x_a coins. A *negative* value of x_a means that Alice should receive $-x_a$ coins from the nanocontract, and, in case of nanocontract closing, Alice *gains* x_a coins in her account in channel γ (the situation of x_b in case of Bob is analogous). Intuitively, $x_a + x_b$ denotes the net amount of coins that is blocked in the nanocontract. If $x_a + x_b = 0$ (or equivalently $x_a = -x_b$) then there are no more coins blocked in the nanocontract, and normally the nanocontract will be terminated. If $x_a > 0$ then as a consequence of this nanocontract Alice "pays" x_a coins to Bob, and if $x_b > 0$ then, symmetrically, Bob "pays" x_b coins to Alice. This way of modeling cash in nanocontracts may look a bit complicated. The reason why it is done this way, is that we need to be ready to handle parallelism, i.e., we have to be prepared for a situation when several nanocontract are executed simultaneously.

Subprotocol for closing the channel

To close the basic payment channel with identifier id a party P that is one of the end-users of the channel with identifier id proceeds as follows:

Procedure `CloseStart`(id) run by party P:

1. Let $\gamma^P := \Gamma^P(id)$ and call a function $\mathsf{close}(\gamma^P)$ in $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$.

Procedure `CloseReact`(id) run by party P':

2. Upon receiving a message closing from $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ let $\gamma^{P'} := \Gamma^{P'}(id)$ and call a function $\mathsf{close}(\gamma^{P'})$ in $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$.
3. Upon receiving a message closed from $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ (together with some coins) remove the channel with identifier id from $\Gamma^{P'}$ and return channel-closed.

Back to procedure `CloseStart`(id) run by party P:

4. After receiving message closing from $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ wait time at most Δ to receive a message closed (with some coins) from $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$. If you do not receive it then call a function finalize() in $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$.
5. Upon receiving a message closed from $\mathit{PaymentContract}_{\gamma.\mathsf{id}}$ (together with some coins) remove the channel with identifier id from $\Gamma^{P'}$ and return channel-closed.

(a)

Function $\mathsf{close}(\gamma^P)$ **called by** $P \in$ end-users

Check if γ^P is correctly signed by $P' := \mathsf{other\text{-}party}(P)$, if $\gamma^P.\mathsf{id} = \mathsf{id}$ and if $\gamma^P.\mathsf{total\text{-}cash} = \mathit{this}.\mathsf{total\text{-}cash}$. If at least one of the above does not hold then go idle. Otherwise consider three cases:

- A variable this.recorded is defined and is equal to $(\widehat{P}, \widehat{\gamma}, \widehat{\tau})$ with $\widehat{P} = P$ — in this case go idle.
- A variable this.recorded is defined and is equal to $(\widehat{P}, \widehat{\gamma}, \widehat{\tau})$ with $\widehat{P} = \mathsf{other\text{-}party}(P)$ — in this case let:

$$\gamma^* := \begin{cases} \gamma^P \text{ if } \gamma^P.\mathsf{ver\text{-}num} \geq \widehat{\gamma}.\mathsf{ver\text{-}num} \\ \widehat{\gamma} \quad \text{otherwise.} \end{cases}$$

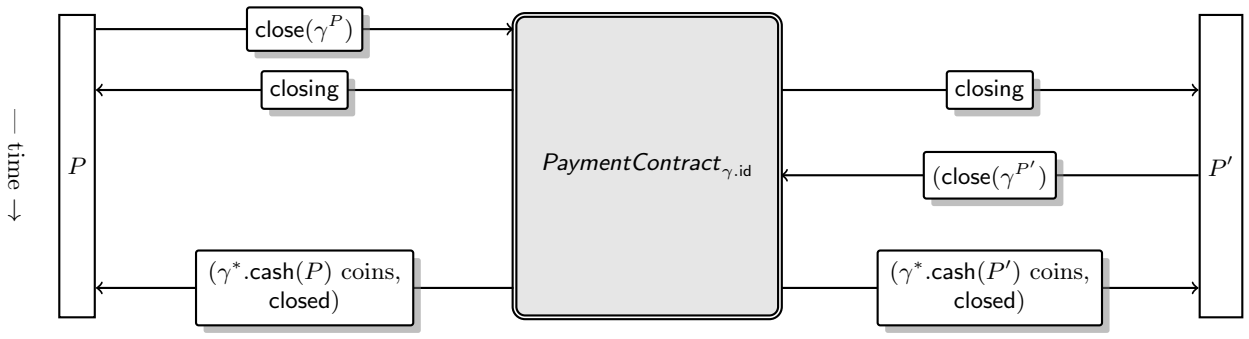
  For each $T \in$ end-users send a message closed together with $\gamma^*.\mathsf{cash}(T)$ coins to T and terminate.
- Otherwise (if this.recorded is undefined) set $\mathit{this}.\mathsf{recorded} := (P, \gamma^P, \mathsf{current\text{-}time})$, send a message closing to both parties in end-users, and go idle.

Function finalize() **called by** $P \in$ end-users

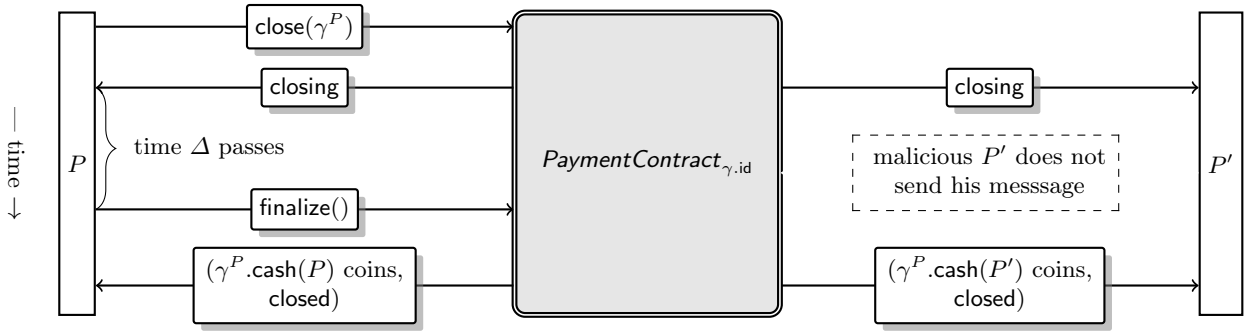
If your storage contains a tuple $(\widehat{P}, \widehat{\gamma}, \widehat{\tau})$, such that $\widehat{\tau} + \Delta <$ current-time then for each $T \in$ end-users send a message closed together with $\widehat{\tau}.\mathsf{cash}(T)$ coins to T and terminate. Otherwise go idle.

(b)

Fig. 9: Closing the channel: (a) the subprotocol, and (b) the corresponding contract functions.



(a)



(b)

Fig. 10: The message flows for closing the channel in case when P' replies (a), and in case P' does not reply (b).

A nanocontract is *active* if its storage is not equal to terminated, and is *inactive* otherwise. We assume that a nanocontract can be made inactive only if the total amount of coins that is blocked in it is zero.

The important thing is that once the parties start to execute the nanocontract, none of them can lose more money than it deposited in the nanocontract. This is because the values y_a and y_b (that are output by Eval) are always non-negative and hence as a result of every step of the execution the values x_a and x_b blocked in the nanocontract can only decrease. Note that this means that the users have to initially block the maximum amount of coins that they can potentially lose in the contract. We emphasize that this is necessary if we want a guarantee that every party will always pay the amount that the nanocontract requires them to pay (see Sect. 4.4 for a discussion on alternative approaches).

As a concrete example consider a nanocontract for tossing two symmetric coins (this paragraph assumes some familiarity with the topic of fair coin tossing over cryptocurrencies, see [2, 3]). Alice and Bob first deposit 2 coins (each) and then they execute a nanocontract, that tosses two coins. After each toss: if the result is "heads" then Alice wins one coin, and if the result is "tails" then Bob wins one coin. Then the possible evolution of the pair (x_a, x_b) looks as on Fig. 11 Note that both

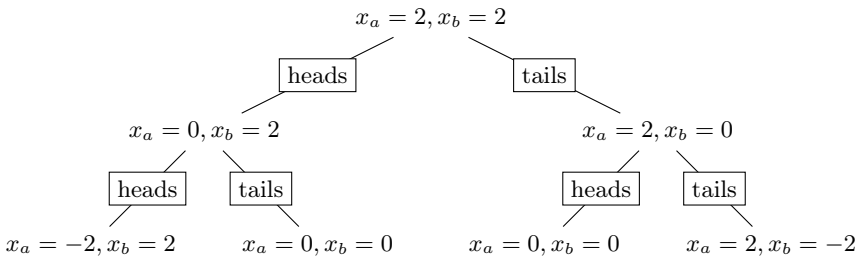


Fig. 11: The tree presenting the evolution of the (x_a, x_b) pair in a nanocontract that tosses two coins (when the result is "heads" Alice wins 1 coin, and otherwise Bob wins 1 coin).

parties need to initially deposit 2 coins (since this is the maximal amount of coins that a party can lose). "Heads" results in decreasing x_a by $y_a = 2$ and "tails" in decreasing x_b by $y_b = 2$. Informally: if the result is "heads" then Alice gets 1 coin from Bob, and 1 coin back from her own blocked deposit (symmetrically for "tails" and Bob). In other words, the output of the Eval function in this case will have a form $(\sigma'; \text{“heads”}, 2; \text{“heads”}, 0)$.

A basic multistate channel can be closed (see Sect. 4.2.7) if it contains no active nanocontracts. This is checked by performing the state registration for every nanocontract. Once the channel is closed, the money in γ.cash is given back to the channel participants.

4.2.1 Overview of the scheme. The basic multistate channels are implemented by a protocol denoted $\texttt{MSChannels}^{C}$ (where C is the bilateral contract code that parametrizes the system). Each multistate channel γ will have its corresponding contract $\mathit{MSContract}^{C}_{\gamma.\mathsf{id}}$ on the ledger that will interact with both γ.Alice and γ.Bob. The names of the functions in $\mathit{MSContract}^{C}_{\gamma.\mathsf{id}}$ are listed on Fig. 12.

A channel is *active* if the corresponding $\mathit{MSContract}^{C}_{\gamma.\mathsf{id}}$ exists on the ledger and it has not been terminated. The general picture is very similar to the one for the payment channels (see Sect. 4.1.2). Each party $P \in \mathcal{P}$ maintains a channel space Γ^{P} that contains all the active basic channels that this party established. *The initial form* of a channel with identifier *id* is equal to γ

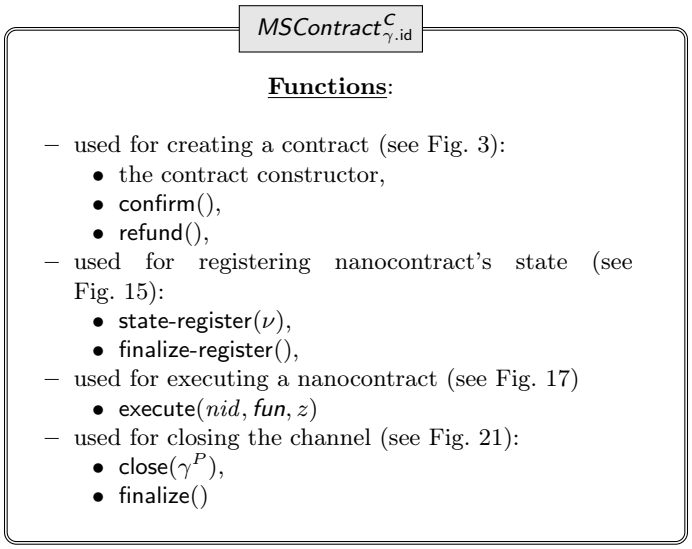


Fig. 12: Functions in the *MSContract*.

with all attributes defined as when the channel was created. For every id the values of attributes $\Gamma^P(id).\mathsf{id}, \Gamma^P(id).\mathsf{Alice}$, and $\Gamma^P(id).\mathsf{Bob}$ will always be the equal to the corresponding values in the initial form of this channel. $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ will be initialized with the initial form of γ, however, unlike in the case of payment channels, γ stored by $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ will evolve. In particular: $\gamma.\mathsf{nspace}$ will initially be empty, and it will get filled-in with nanocontracts as a result of the "nanocontract state registration" procedure (see Sect. 4.2.4).

The general structure of our system, including the message types that are sent between the parties and the contract, are depicted on Fig. 13.

4.2.2 Maintaining channel state. Maintaining the state of the multistate channels without touching the ledger is a little bit more tricky than for the payment channels (see Sect. 4.1). This is because a channel can contain several nanocontracts, that can be created and updated in parallel. At the first sight it may look natural to extend the idea from Sect. 4.1 by letting every P add to γ two attributes $\gamma.\mathsf{ver\text{-}num}$ and $\gamma.\mathsf{sign}$, where where $\gamma.\mathsf{ver\text{-}num} \in \mathbb{N}$ is a counter that is incremented with every channel update and $\gamma.\mathsf{sign}$ is a signature of $P' = \gamma.\mathsf{other\text{-}party}(P)$. Unfortunately, for reasons explained in Sect. 4.2.9, we need a more fine-grained control over the state of the nanocontracts in the channel.

Our solution is as follows. The parties will maintain an independent version number of each nanocontract in $\gamma.\mathsf{nspace}$. Concretely, for each nanocontract we add the attributes ver-num and sign that are internally used by the parties for the protocol's execution. The counter $\nu.\mathsf{ver\text{-}num} \in \mathbb{N}$ is incremented with every nanocontract update (see Sect. 4.2.6) and $\nu.\mathsf{sign}$ is a signature of $P' = \gamma.\mathsf{other\text{-}party}(P)$ on $(\nu.\mathsf{nid}, \nu.\mathsf{blocked}(\gamma.\mathsf{Alice},), \nu.\mathsf{blocked}(\gamma.\mathsf{Bob}), \nu.\mathsf{storage}, \nu.\mathsf{ver\text{-}num})$. Hence, every tuple $\nu \in \gamma.\mathsf{nspace}$ has the following form (cf. Eq. (3)):

$$(\nu.\mathsf{nid}, \nu.\mathsf{blocked}, \nu.\mathsf{storage}, \nu.\mathsf{ver\text{-}num}, \nu.\mathsf{sign}).$$

If $\nu.\mathsf{sign}$ is a correct signature of P on $(\nu.\mathsf{nid}, \nu.\mathsf{blocked}(\gamma.\mathsf{Alice}), \nu.\mathsf{storage}, \nu.\mathsf{blocked}(\gamma.\mathsf{Bob}), \nu.\mathsf{ver\text{-}num})$ then we say that it is *a version of a state of a nanocontract (with identifier* $\nu.\mathsf{nid}$*) signed by* P. The role of the attributes ver-num and sign is similar to their role in the basic payment channel. They are "private" and can only be accessed by the parties locally during the protocols execution. The value

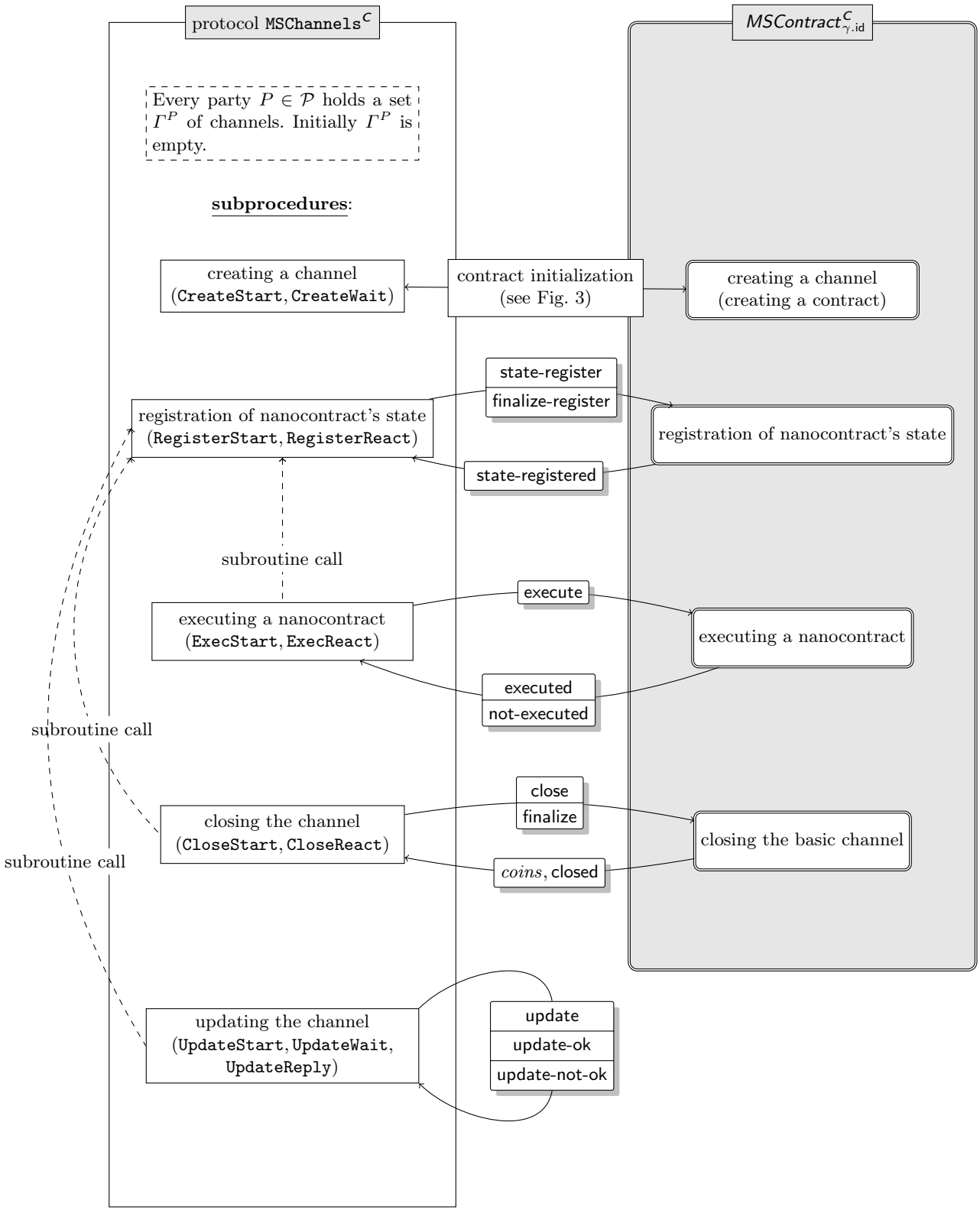


Fig. 13: The general structure of the scheme for basic multistate channels, including the messages that are sent between the parties and the contract, and the names of the procedures.

ν.ver-num denotes a counter indicating the latest version number of ν that a party P has stored locally. The attribute ν.sign, on the other hand, is a signature of P' on (ν.nid, ν.blocked, ν.ver-num). Jumping a little bit ahead, ν.ver-num will be incremented by 1 in every nanocontract update (see Sect. 4.2.6), and in case of a dispute between Alice and Bob ν will be considered to be the valid one if it has the highest version number.

4.2.3 Creating a channel. The subprotocol for creating a multistate channel γ consists of procedures `CreateStart` (executed by the party that initiates the subprotocol, which in our case will always be γ.Alice), and `CreateStart` (executed by the other party, γ.Bob). Both procedures get γ as input. We assume that the channel γ has γ.nspace $= \emptyset$ (in other words: there are no nanocontracts in the "default" state of the channel). The procedure is very similar to the creation procedure for the payment channels (see Sect. 4.1.3). It is depicted on Fig. 14. Essentially, the only difference is that in this case we do not define the γ.ver-num attribute. This is because the (extended) multistate channels do not have the .ver-num attribute. Instead, as described in Sect. 4.2.2, this attribute is placed in the nanocontract (and will be set in the "update" procedure, see Sect. 4.2.6). The only other technical change is that the contract is now called *MSContract*.

At any point in time parties P_a and P_b can decide to create a multistate channel γ such that γ.Alice $= P_a$ and γ.Bob $= P_b$ and channel γ has γ.nspace $= \emptyset$ (in other words: we require that there are no nanocontracts in the "default" state of the channel). This can be done only if no channel with identifier γ.id has been created before, and if γ.Alice and γ.Bob have $x_a := \gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ and $x_b := \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ unspent coins on the ledger (respectively). We assume that the parties agree beforehand on all the attributes of γ and on the exact round τ when the creation procedure starts. The subprotocol for creating a multistate channel γ consists of procedures `CreateStart` (executed by the party that initiates the subprotocol, which in our case will always be γ.Alice), and `CreateWait` (executed by the other party, γ.Bob). Both procedures get γ as input. To create γ parties γ.Alice and γ.Bob simply run the protocol for initializing *MSContract* (see Sect. 3.2) with parameter $\pi := \gamma$. Since the response time of ledger contracts is at most Δ time, the standard execution time is $2 \cdot \Delta$ and pessimistic execution time of this procedure is $3 \cdot \Delta$ (because γ.Alice may need to use refund()).

Subprotocol for creating a channel

Procedure `CreateStart`(γ) run by γ.Alice

1. Let $w_a :=$ `InitializeStart`(*MSContract*; γ).
2. If $w_a =$ initialized then store γ in Γ and output created.
3. Otherwise (if $w_a =$ not-initialized) output not-created.

Procedure `CreateWait`(γ) run by γ.Bob

1. Let $w_b :=$ `InitializeWait`(*MSContract*; γ).
2. If $w_b =$ initialized then store γ in Γ and output created.
3. Otherwise (if $w_b =$ not-initialized) output not-created.

Fig. 14: The "Multistate channel creation" subprocedure

4.2.4 Registration of the state of a nanocontract. The goal of the "nanocontract state registration" procedure is to convince the $\mathit{MSContract}^{C}_{\gamma.\mathsf{id}}$ about the current state of the a nanocontract in the channel. This is done in a manner similar to the one for determining the current state of the payment channel (used in the closing procedure, see Sect. 4.1.5), except that the procedure is

performed independently for every nanocontract. This procedure will be executed in two cases. The first one is when Alice and Bob enter into a disagreement about the state of some nanocontract (during the execution of nanocontract update procedure, or at the beginning of the nanocontract execution procedure, see Sections 4.2.6 and 4.2.5, resp.), which happens only if one of them is corrupt. The second one is when they want to close the channel with to which the nanocontract belongs (in which case it will be performed for every nanocontract in this channel, see Sect. 4.2.7). Let id be the identifier of a channel between Alice and Bob, and let nid be the identifier of a nanocontract in this channel. To register the state of this nanocontract each party sends to $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ her latest versions of the nanocontract with identifier nid signed by the other party, and the nanocontract with the higher ver-num attribute "wins", i.e., is considered by $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ to be the one that counts.

Going a bit more into technical details, the state registration procedure works as follows. Since $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ cannot take any actions by itself, it has to be "woken up" by a message from a user $P \in \gamma.\mathsf{end\text{-}users}$ who wants to register the state (since he thinks that the other user turned malicious, or he wants to close the channel). This will be done by calling a function $\mathsf{state\text{-}register}(\nu^P)$ in $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$, were $\nu^P = \Gamma^P(id).\mathsf{nspace}(nid)$.

After receiving such an initialization call, $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ notifies both parties P and P' that it received a call $\mathsf{state\text{-}register}(\nu^P)$ and waits time Δ for a function call $\mathsf{state\text{-}register}(\nu^{P'})$ from party P' (where $\nu^{P'} = \Gamma^{P'}(id).\mathsf{nspace}(nid)$). After receiving this call the contract decides what is the current state of the nanocontract by taking ν with a higher version number. More precisely, it lets:

$$\nu^* := \begin{cases} (\nu^P.\mathsf{nid}, \nu^P.\mathsf{blocked}, \nu^P.\mathsf{storage}) & \text{if } \nu^P.\mathsf{ver\text{-}num} \geq \\ & \nu^{P'}.\mathsf{ver\text{-}num} \\ (\nu^{P'}.\mathsf{nid}, \nu^{P'}.\mathsf{blocked}, \nu^{P'}.\mathsf{storage}) & \text{otherwise,} \end{cases}$$

stores ν^* as the value of $\mathsf{nspace}(nid)$, and sends a confirmation message $(\mathsf{state\text{-}registered}, \nu^*)$ to both parties (we now say that *the state of the nanocontract with identifier nid is registered*). Note that we do not define the values of the ver-num and sign attributes, since once the state of a nanocontract is registered they are not needed anymore.

If the state-register call is not received from P' within time Δ, then only the call from P counts (hence $\nu^* := \nu^P$). Since the contract never does anything "by itself", P has to wake up $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ after this time passed by calling a "wake up" function finalize-register.

The state registration procedure is depicted on Fig. 15 and the message flow in both scenarios (when P replied to the contract, and when it did not) is depicted on Fig. 16.

We assume that once the state of a nanocontract has been registered, this nanocontract will never be updated by the users via the update procedure (see Sect. 4.2.6). This can be done without loss of generality, since the state registration procedure will be executed only if Alice and Bob are in disagreement (or if they are closing the channel), and the updates are performed only between the parties that do not have any disagreements between each other. Once the channel state is registered, the only changes in the channel state can now come from performing the "execution" procedure, described below.

Let us finally provide some details about the timing about the registration of the state of a nanocontract. The pessimistic execution time of this procedure is $3 \cdot \Delta$, since in the worst case the procedure takes 3 rounds of interaction with the ledger: (a) P sends a message to a ledger, (b) P' has time Δ to reply, (c) if P' did not reply then P sends the finalize-register message. The standard execution time is $2 \cdot \Delta$ (since Step (c) is not needed).

4.2.5 Execution of a nanocontract. Some general intuitions behind the nanocontract execution were already provided at the beginning of Sect. 4.2. For the sake of simplicity we do not allow the

Registration of the state of a nanocontract:

To register the state of the nanocontract with identifier nid in the basic multistate channel with identifier id, a party P (that is one of the end-users of this channel) proceeds as follows:

Procedure `RegisterStart`(id, nid) run by party P:

1. Let $\nu^P := \Gamma^P(id).\mathsf{nspace}(nid)$ and call a function $\mathsf{state\text{-}register}(nid, \nu^P)$ in $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$.

Procedure `RegisterReact`(id, nid) run by party P':

2. Upon receiving a message $(\mathsf{state\text{-}registering}, nid)$ from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ let $\nu^{P'} := \Gamma^{P'}(id).\mathsf{nspace}(nid)$ and call a function $\mathsf{state\text{-}register}(nid, \nu^{P'})$ in $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$.
3. Upon receiving a message $(\mathsf{state\text{-}registered}, \nu^*)$ from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ let $\Gamma^{P'}(id).\mathsf{nspace}(nid) := \nu^*$ and return $\mathsf{established}(\nu^*)$.

Back to procedure `RegisterStart`(id, nid) run by party P:

4. After receiving message $\mathsf{state\text{-}registering}$ from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ wait time at most Δ to receive a message $(\mathsf{state\text{-}registered}, \nu^*)$ from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$. If you do not receive it, then call a function $\mathsf{finalize\text{-}register}()$ in $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$.
5. Upon receiving $(\mathsf{state\text{-}registered}, \nu^*)$ let $\Gamma^P(id).\mathsf{nspace}(nid) := \nu^*$ and return $\mathsf{established}(\nu^*)$.

(a)

Function $\mathsf{state\text{-}register}(\nu)$ **called by** $P \in \mathsf{end\text{-}users}$

Check if ν is correctly signed by $P' := \mathsf{other\text{-}party}(P)$. If not, then ignore this call. Otherwise consider two cases:

- Your storage contains a value $(\widehat{P}, \widehat{nid}, \widehat{\nu}, \widehat{\tau})$ with $\widehat{P} = P$ — in this case ignore this call.
- Your storage contains a value $(\widehat{P}, \widehat{nid}, \widehat{\nu}, \widehat{\tau})$ with $\widehat{P} = \mathsf{other\text{-}party}(P)$ and $\widehat{nid} = \nu.\mathsf{nid}$ — in this case let:

$$\nu^* := \begin{cases} (\nu.\mathsf{nid}, \nu.\mathsf{blocked}, \nu.\mathsf{storage}) & \text{if } \nu.\mathsf{ver\text{-}num} \geq \widehat{\nu}.\mathsf{ver\text{-}num} \\ (\widehat{\nu}.\mathsf{nid}, \widehat{\nu}.\mathsf{blocked}, \widehat{\nu}.\mathsf{storage}) & \text{otherwise.} \end{cases}$$

  Let $\mathsf{nspace}(nid) := \nu^*$, erase $(\widehat{P}, \widehat{nid}, \widehat{\nu}, \widehat{\tau})$ from your storage, send a message $(\mathsf{state\text{-}registered}, \nu^*)$ to both parties in $\mathsf{end\text{-}users}$ and go idle.
- Otherwise store $(P, nid, \nu, \mathsf{current\text{-}time})$ in your storage, send a message $\mathsf{state\text{-}registering}(nid)$ to both parties in $\mathsf{end\text{-}users}$, and go idle.

Function $\mathsf{finalize\text{-}register}()$ **called by** $P \in \mathsf{end\text{-}users}$

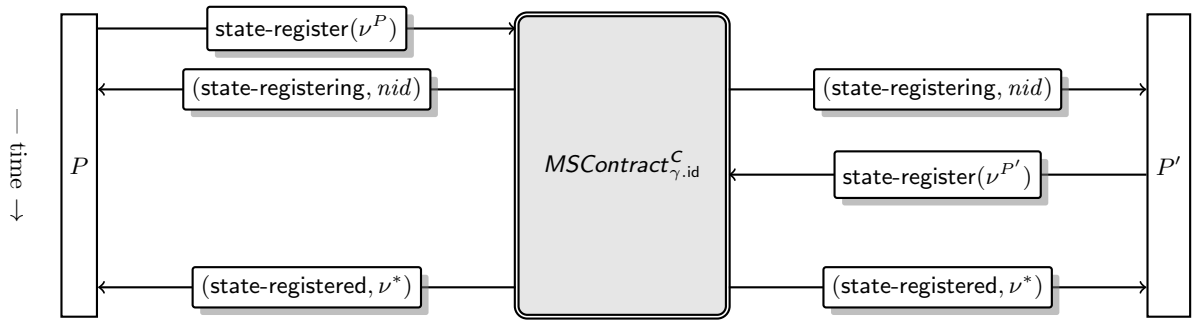
If your storage contains a value $(\widehat{P}, \widehat{nid}, \widehat{\nu}, \widehat{\tau})$ such that $\widehat{P} = P$, $\widehat{nid} = \nu.\mathsf{nid}$, and $\tau + \Delta < \mathsf{current\text{-}time}$ then set

$$\nu^* := (\widehat{\nu}.\mathsf{nid}, \widehat{\nu}.\mathsf{blocked}, \widehat{\nu}.\mathsf{storage}).$$

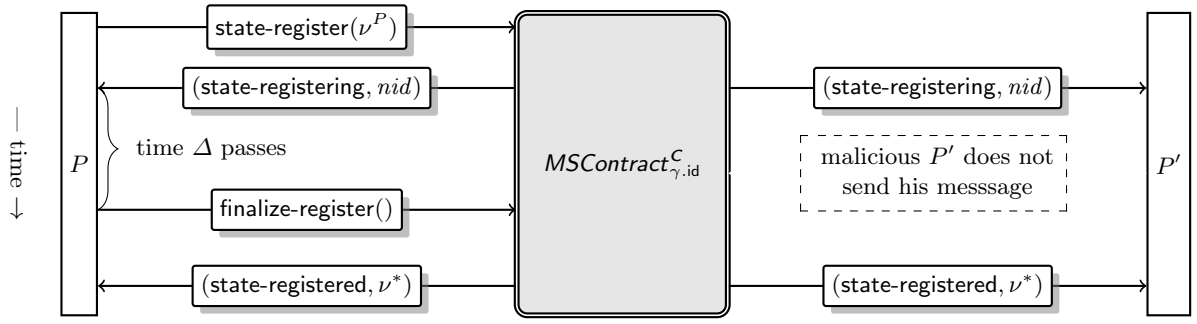
Let $\mathsf{nspace}(nid) := \nu^*$, erase $(\widehat{P}, \widehat{nid}, \widehat{\nu}, \widehat{\tau})$ from your storage, send a message $(\mathsf{state\text{-}registered}, \nu^*)$ to both parties in $\mathsf{end\text{-}users}$ and go idle.

(b)

Fig. 15: The state registration procedure: (a) the subprotocol, and (b) the corresponding functions in the contract.



(a)



(b)

Fig. 16: The message flows for registration of nanocontract's state in case when P' replies (a), and in case P' does not reply (b).

parties to add coins when the nanocontract is executed, i.e., we will only allow calls to Eval where $x = 0$ (cf. (1)). That is, once the nanocontract has been put on the ledger, the users cannot add coins to it anymore. Instead, the nanocontract can only release coins to the deposits in the basic multistate channel γ (see Sect. 4.2.10 for explanations of the reasons for this, and an outline of a construction that does not have this limitation.

Let us not provide some more details about the execution procedure. Let $\nu := \gamma.\mathsf{nspace}(nid)$ be some nanocontract in channel γ, and let $(x_a, x_b) := (\nu.\mathsf{blocked}(\gamma.\mathsf{Alice}), \nu.\mathsf{blocked}(\gamma.\mathsf{Bob}))$. Then, each end user of γ can execute a function fun in C with parameters z and x coins starting from storage $\sigma = \nu.\mathsf{storage}$. The $\mathit{MSContract}$ will compute $\beta := \mathsf{Eval}^{\mathsf{C}}(\sigma; P, 0, \tau; \mathit{fun}, z)$ (where τ is the current time, and "0" corresponds to the fact that we do not allow the parties to attach any coins when nanocontracts are forcefully executed). If $\beta = \bot$ then nothing happens[16]; otherwise, parse β as $(\sigma', (m_a, y_a), (m_b, y_b))$. As a result of the "channel execution", the new state of the nanocontract is $(x_a - y_a, x_b - y_b, \sigma')$ (i.e.: $y_a + y_b$ coins get "unblocked" in the nanocontract) and Alice and Bob gain y_a and y_b coins (respectively) in their deposits in channel γ, and σ' becomes the new storage of the nanocontract. Moreover Alice and Bob receive messages m_a and m_b (respectively).

Technically, execution of a nanocontract is done in the following way (see also Fig. 17). First, before any computation takes place, the contract $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ needs to understand what is the current state of the nanocontract. To this end, the initiating party P performs the "state registration" procedure described in Sect. 4.2.4. After the state of the nanocontract is registered the contract $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ will accept function calls $\mathsf{execute}(nid, \mathit{fun}, z)$ from $P \in \gamma.\mathsf{end\text{-}users}$. Once $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ such a call, the it computes $(\sigma', (m_a, y_a), (m_b, y_b)) := \mathsf{Eval}^{\mathsf{C}}(\sigma; P, 0, \mathsf{current\text{-}time}; \mathit{fun}, z)$. He then defines ν' as

$$\begin{aligned}
\nu'.\mathsf{nid} &:= nid \\
\nu'.\mathsf{blocked}(\gamma.\mathsf{Alice}) &:= \nu.\mathsf{blocked}(\gamma.\mathsf{Alice}) - y_a \\
\nu'.\mathsf{blocked}(\gamma.\mathsf{Bob}) &:= \nu.\mathsf{blocked}(\gamma.\mathsf{Bob}) - y_b \\
\nu'.\mathsf{storage} &:= \sigma'.
\end{aligned}$$

Next, the contract sends ν' to both $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ together with messages m_a and m_b respectively. Finally, $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ updates the state of the nanocontract as $\gamma.\mathsf{nspace}(nid) := \nu'$.

The above description assumes that P and $P' = \gamma.\mathsf{other\text{-}party}(P)$, behave honestly, in the sense that its P that sends the $\mathsf{execute}$ message to the $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ after the state registration of the nanocontract has been completed. In general, however, the parties can behave arbitrarily, and hence we also accept that $\mathsf{execute}$ is sent by P' (and reaches the contract before the $\mathsf{execute}$ message sent by P). We accept such a situation as correct behavior of the system (such issues are discussed in more detail in Sect. 4.2.9). Another issue that may happen when parties do not follow the protocol is that neither P nor P' sends the $\mathsf{execute}$ message. This is not a problem, since in this case the result is simply that the state of the nanocontract is registered, but no execution was performed. Notice that of course in the latter case parties can execute the nanocontract at some later points in time.

The execution of a nanocontract can be performed multiple times. The only difference between the first execution and all subsequent executions is that in the latter case there is no need to perform the state registration procedure (since the state of a nanocontract is already registered). The details of the protocol and the part of the $\mathit{MSContract}$ contract that corresponds to the forceful execution are shown on Figure 18.

[16] Recall that the role of the $\bot$ output is to indicate that the input α was incorrect. For example this will be output if the message *msg* is sent by Alice, while its "Bob's turn" to send a message, or if *msg* is wrongly formatted.

Nanocontract execution

To execute a nanocontract with identifier nid in the basic multistate channel γ with identifier id, a party P (that is one of the end-users of this channel) proceeds as follows (let $P' := \gamma.\mathsf{other\text{-}party}(P)$):

Procedure $\texttt{ExecStart}(id, nid, \mathit{fun}, z)$ for party P

1. If you have not yet run the `RegisterStart` or `RegisterReact` procedure with parameters (id, nid) then start the $\texttt{RegisterStart}(id, nid)$ procedure and wait until its finished.

Procedure $\texttt{ExecReact}(id, nid)$ for party P'

2. If P initiates the $\texttt{RegisterStart}(id, nid)$ procedure then P' engages in it (i.e., it executes $\texttt{RegisterStart}(id, nid)$.

Back to procedure $\texttt{ExecStart}(id, nid, \mathit{fun}, z)$ for party P

3. Call a function $\mathsf{execute}(nid, \mathit{fun}, z)$ in $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$.

Rest of the procedures $\texttt{ExecStart}(id, nid, \mathit{fun}, z)$ and $\texttt{ExecReact}(id, nid)$ for $T = P$ and $T = P'$ (respectively):

4. Upon receiving a message $(\mathsf{executed}, \nu', m)$ from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$, party T does the following (below let $\gamma^T := \Gamma^T(id)$):
   - $\gamma^T.\mathsf{cash}(\gamma^T.\mathsf{Alice}) := \gamma^T.\mathsf{cash}(\gamma.\mathsf{Alice}) + \nu'(\gamma.\mathsf{Alice})$,
   - $\gamma^T.\mathsf{cash}(\gamma^T.\mathsf{Bob}) := \gamma^T.\mathsf{cash}(\gamma.\mathsf{Bob}) + \nu'(\gamma.\mathsf{Bob})$,
   - $\gamma^T.\mathsf{nspace}(nid) := \nu'$.
   - Return m.

Function $\mathsf{execute}(nid, \mathit{fun}, z)$ **called by** $P \in \mathsf{end\text{-}users}$

If the state of the nanocontract with identifier nid has not been registered or if $\mathsf{nspace}(nid).\mathsf{storage} = \mathsf{terminated}$, then send a message $\mathsf{not\text{-}executed}$ to P and go idle. Otherwise proceed as follows:

1. Let:
   $\nu := \mathsf{nspace}(nid)$,
   $x_a := \nu.\mathsf{blocked}(\mathsf{Alice})$, and
   $x_b := \nu.\mathsf{blocked}(\mathsf{Bob})$
2. Let $\beta = \mathsf{Eval}^{\mathcal{C}}(\nu.\mathsf{storage}; P, 0, \mathsf{current\text{-}time}; \mathit{fun}, z)$. If $\beta = \mathsf{ignore}$ then ignore this call.
   Otherwise set $(\sigma, (m_a, y_a), (m_b, y_b)) := \beta$. If $y_a + y_b > x_a + x_b$ then stop without sending any message. Otherwise:
   (a) Set:
   $\nu'.\mathsf{blocked}(\mathsf{Alice}) := x_a - y_a$,
   $\nu'.\mathsf{blocked}(\mathsf{Bob}) := x_b - y_b$, and
   $\nu'.\mathsf{storage} := \sigma$.
   (b) Let $\mathsf{nspace}(nid) := \nu'$.
   (c) Send $(\mathsf{executed}, \nu', m_a)$ and $(\mathsf{executed}, \nu', m_b)$ to Alice and Bob respectively.

Fig. 17: The nanocontract execution procedure and contract.

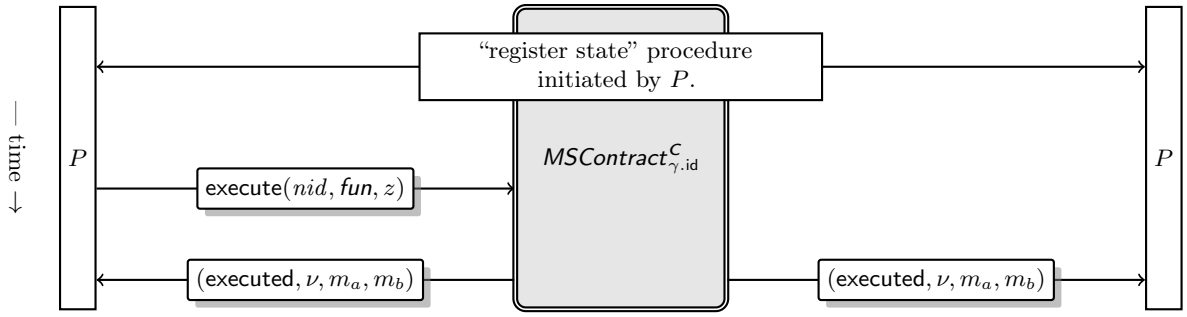


Fig. 18: Message flow for the nanocontract execution procedure.

Finally, let us discuss the execution time of the nanocontract execution procedure. In the pessimistic case, a execution will take $4 \cdot \Delta$ rounds, where $3 \cdot \Delta$ rounds come from the pessimistic execution time of the state registration procedure, and Δ from the time that is needed to run the nanocontract execution.

4.2.6 Updating the channel. As in the case of the basic payment channels, we allow the end-users of the basic multistate channel to update the nanocontract's state multiple times without touching the blockchain. This procedure will also be used to create nanocontracts (see Sect. 4.2.8). Conceptually, this is done in a similar way as for the basic payment channels. Namely, to update the state of nanocontract nid (or to create it, if it does not exist) with values $(\tilde{x}_a, \tilde{x}_b, \tilde{\sigma})$ the parties $P, P' \in \gamma.\mathsf{end\text{-}users}$ proceed in the three round protocol shown in Figure 19 (we again assume that the parties agreed beforehand that they want to start this subprotocol). We give further explanation of the protocol execution below.

The updating procedures take an additional parameter $\Upsilon \in \mathsf{time}$ that denotes the time that the non-initiating party (P') has to decide if she accepts the proposed update, or not. This is needed for constructing higher-level protocols such as our virtual payment channels from Section 4.3. In this protocol we will need to allow party P' to abort the update procedure initiated by party P, without resulting into a situation where the state of the nanocontract nid is registered on the ledger.

Party P, who initiates the update, starts procedure $\texttt{UpdateStart}(id, nid, \tilde{\sigma}, \tilde{x}_a, \tilde{x}_b, \Upsilon)$, while P' starts procedure $\texttt{UpdateWait}(id, nid, \tilde{\sigma}, \tilde{x}_a, \tilde{x}_b, \Upsilon)$. If the protocol succeeds, then the parties update their local information about their deposits in the channel with identifier id, i.e., both $T \in \Gamma(id).\mathsf{end\text{-}users}$ let $\gamma^T := \Gamma^T(id)$ and $\nu^T := \gamma^T.\mathsf{nspace}(nid)$, and then:

$$\begin{aligned}
\Gamma^T(id).\mathsf{cash}(\gamma^T.\mathsf{Alice}) &:= \gamma^T.\mathsf{cash}(\gamma^T.\mathsf{Alice}) \\
&\quad + \nu^T.\mathsf{blocked}(\gamma^T.\mathsf{Alice}) - \tilde{x}_a, \\
\Gamma^T(id).\mathsf{cash}(\gamma^T.\mathsf{Bob}) &:= \gamma^T.\mathsf{cash}(\gamma^T.\mathsf{Bob}) \\
&\quad + \nu^T.\mathsf{blocked}(\gamma^T.\mathsf{Bob}) - \tilde{x}_b.
\end{aligned}$$

Moreover, the state of the nanocontract with identifier nid held by party T changes in the following way (below where $T' := \gamma^T.\mathsf{other\text{-}party}(T)$):

$$\begin{aligned}
&\Gamma^T(id).\mathsf{nspace}(nid).\mathsf{blocked}(\gamma^T.\mathsf{Alice}) := \tilde{x}_a \\
&\Gamma^T(id).\mathsf{nspace}(nid).\mathsf{blocked}(\gamma^T.\mathsf{Bob}) := \tilde{x}_b \\
&\Gamma^T(id).\mathsf{nspace}(nid).\mathsf{storage} := \tilde{\sigma} \\
&\Gamma^T(id).\mathsf{nspace}(nid).\mathsf{ver\text{-}num} := \nu^T.\mathsf{ver\text{-}num} + 1 \\
&\Gamma^T(id).\mathsf{nspace}(nid).\mathsf{sign} := \mathsf{Sign}_{\mathsf{sk}_{T'}}(\nu),
\end{aligned} \tag{5}$$

Let us now describe the nanocontract update protocol in more detail. To initiate the update of the nanocontract with identifier nid, party P runs a procedure denoted $\texttt{UpdateStart}(id, nid, \tilde{\sigma}, \tilde{x}_a, \tilde{x}_b, \Upsilon)$, and runs a procedure $P' := \gamma^P.\mathsf{other\text{-}party}(P)$ denoted $\texttt{UpdateWait}(id, nid, \tilde{\sigma}, \tilde{x}_a, \tilde{x}_b, \Upsilon)$. As the first step of her procedure, party P sends a message update with a signature of the new state to P', i.e.,

$$\mathsf{Sign}_{\mathsf{sk}_P}(\tilde{x}_a, \tilde{x}_b, \tilde{\sigma}, \nu^P.\mathsf{ver\text{-}num} + 1).$$

If P' receives such a message within one round, then P' has to react to it by calling a procedure denoted $\texttt{UpdateReply}$ within time at most Υ, and either agreeing for the update, or not. More precisely P' has to following options.

1. P' calls `UpdateReply`(ok, id, nid) by sending an update-ok message along with the signature
$$\mathsf{Sign}_{\mathsf{sk}_{P'}}(\tilde{x}_a, \tilde{x}_b, \tilde{\sigma}, \nu^{P'}.\mathsf{ver\text{-}num} + 1)$$
to party P. This results in updating the local state of the parties P and P' and successfully terminating the nanocontract update.
2. P' calls `UpdateReply`(not-ok, id, nid): P' reverts to the state $\nu^{P'}$ that the nanocontract with identifier nid has held before the update was started. Technically, this is done by P' by sending to P a update-not-ok message with the following signature:
$$\mathsf{Sign}_{\mathsf{sk}_{P'}}(\nu^{P'}.\mathsf{blocked}(\gamma^{P'}.\mathsf{Alice}), \nu^{P'}.\mathsf{blocked}(\gamma^{P'}.\mathsf{Bob}), \nu^{P'}.\mathsf{storage}, \nu^{P'}.\mathsf{ver\text{-}num} + 2),$$
where the only change compare to $\nu^{P'}$ is the higher version number (i.e.,$\nu^{P'}.\mathsf{ver\text{-}num} + 2$ instead of just $\nu^{T'}.\mathsf{ver\text{-}num}$). Notice that P' need to add 2 to the version number, because $\nu^{P'}.\mathsf{ver\text{-}num}+1$ has already been used when P initiated the nanocontract update.

In the next phase of the protocol `UpdateStart` executed by P is activated within time at most $\Upsilon + 1$ by one of the following three events.

1. It received a update-ok message, in which case it will update its local state to the new state from Eq. (5).
2. It received a update-not-ok message, in which case it will revert to the state before the update was called, but now with a higher version number. Notice that in this case P will also send a signature of the state with higher version number to P'.
3. Time $\Upsilon + 1$ has passed and non of the above messages has been received. In this case, P' is corrupt and P will establish its state using the `RegisterStart`(id, nid) subprocedure.

In the optimistic case when the parties follow the protocol, then the execution time is upper bounded by $\Upsilon + 3$. Υ comes from the fact that we give party P' additionally time Υ to react to an update message received by P. Moreover, if P' calls the procedure `UpdateReply` with $z =$ not-ok, then the parties need additionally 3 rounds to complete the protocol. In the pessimistic case, when one of the parties does not follow the protocol, then we may need to register the state of the nanocontract on the ledger, which requires additionally $3 \cdot \Delta$ rounds. Hence, in the pessimistic case the maximal running time of the protocol is $\Upsilon + 3 \cdot \Delta + 3$ rounds.

4.2.7 Closing the basic multistate channel. At some point in time, one of the parties may want to close the channel γ (to "convert" her money x_P to real coins on the ledger). We will assume that a contract $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ will allow any party $P \in \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$ to close the channel if it does not contain any active nanocontracts. Recall that a nanocontract ν is called active, when $\nu.\mathsf{total\text{-}blocked} > 0$. If the parties do not have any dispute then they can simply update the state of the channel in such a way that it does not contain a nanocontract with non-zero amount of coins blocked (via the update procedure from Sect. 4.2.6).

What is a little bit more tricky is handling the situation when the parties are in disagreement. Note that in principle there can exists a nanocontract ν in γ such that a party P is not able to "execute" ν in such a way that all the money from x is removed (for example: ν is waiting for a message from $\gamma.\mathsf{other\text{-}party}(P)$, and P has no way to "push" it forward). Hence, P may end up with a channel that is impossible to close. We solve this problem in the following way: technically, we allow such a situation to happen, and we say that it is up to the users to choose the specification of the contract C such that each party can always close the nanocontract.[17]

[17] For example, C can accept a special message "kill" such that if kill is sent by any party P after some time τ' (determined in σ) then the money from the nanocontract is distributed in some specified way, and the nanocontract is emptied.

Subprotocol to update a nanocontract nid in basic multistate channel with identifier id

Let P be one of the end-users of the basic multistate channel with identifier id. For $T \in \gamma.\mathsf{end\text{-}users}$ let $\gamma^T := \Gamma^T(id)$ be T's local version of the basic multistate channel with identifier id and define the following abbreviations:

- $\nu^T := \gamma^T.\mathsf{nspace}(nid)$.
- $x_a^T := \nu^T.\mathsf{blocked}(\gamma^T.\mathsf{Alice})$ and $x_b^T := \nu^T.\mathsf{blocked}(\gamma^T.\mathsf{Bob})$ (if ν^T is undefined, then set $x_a^T = 0$ and $x_b^T = 0$).
- $\sigma^T := \nu^T.\mathsf{storage}$ (if ν^T is undefined, then set $\sigma^T := \bot$.)
- $ver\text{-}num^T := \nu^T.\mathsf{ver\text{-}num}$.

The protocol is run between the two end-users P and P' of the channel with identifier id. It enables the end-users to update the nanocontract with identifier nid with values $(\tilde{\sigma}, \tilde{x}_a, \tilde{x}_b)$ and consists of multiple procedures defined below.

Procedure $\texttt{UpdateStart}(id, nid, \tilde{\sigma}, \tilde{x}_a, \tilde{x}_b, \Upsilon)$ run by P:

1. Compute the signature
$$s^P := \mathsf{Sign}_{\mathsf{sk}_P}(nid, \tilde{x}_a, \tilde{x}_b, \tilde{\sigma}, ver\text{-}num^P + 1),$$
and send (update, s^P) to $P' := \gamma^P.\mathsf{other\text{-}party}(P)$. Wait for at most $\Upsilon + 1$ rounds.

Procedure $\texttt{UpdateWait}(id, nid, \tilde{\sigma}, \tilde{x}_a, \tilde{x}_b, \Upsilon)$ run by P':

Upon receiving a message (update, s^P) from party P, party P' proceeds as follows:

2. If s_P is a signature of the following form:
$$\mathsf{Sign}_{\mathsf{sk}_P}(nid, \tilde{x}_a, \tilde{x}_b, \tilde{\sigma}, ver\text{-}num^{P'} + 1),$$
then return $\mathsf{update\text{-}requested}$ and wait at most until round Υ for a call to the procedure $\texttt{UpdateReply}$ defined below.

Procedure $\texttt{UpdateReply}(z, id, nid)$ run by P':

If $z = \mathsf{ok}$ then execute the following:

3. Update local version of nanocontract with $\Gamma^{P'}(id).\mathsf{nspace}(nid) := (\tilde{x}_a, \tilde{x}_b, \tilde{\sigma}, ver\text{-}num^{P'} + 1, s^P)$.
4. Set $\Gamma^{P'}(id).\mathsf{cash}(\gamma^{P'}.\mathsf{Alice}) := \gamma^{P'}.\mathsf{cash}(\gamma^{P'}.\mathsf{Alice}) - \tilde{x}_a$ and $\Gamma^{P'}(id).\mathsf{cash}(\gamma^{P'}.\mathsf{Bob}) := \gamma^{P'}.\mathsf{cash}(\gamma^{P'}.\mathsf{Bob}) - \tilde{x}_b$.
5. Compute the following valid signature:
$$s^{P'} := \mathsf{Sign}_{\mathsf{sk}_{P'}}(nid, \tilde{x}_a, \tilde{x}_b, \tilde{\sigma}, ver\text{-}num^{P'} + 1)$$
and send $(\mathsf{update\text{-}ok}, s^{P'})$ to P and return $\mathsf{updated}$.

Otherwise (if $z = \mathsf{not\text{-}ok}$) then execute the following:

6. Compute a valid signature
$$s^{P'} := \mathsf{Sign}_{\mathsf{sk}_{P'}}(nid, x_a^{P'}, x_b^{P'}, \sigma^{P'}, ver\text{-}num^{P'} + 2),$$
and send the message $(\mathsf{update\text{-}not\text{-}ok}, s^{P'})$ to party P and wait 1 round.

Procedure for updating nanocontract (continued on the next page).

Subprotocol to update a nanocontract nid in basic multistate channel with identifier id (continued)

Back to procedure $\texttt{UpdateStart}(id, nid, \tilde{\sigma}, \tilde{x}_a, \tilde{x}_b, \Upsilon)$ run by P:

Depending on whether one of the following events happen, party P wakes up and proceeds as follows:

7. Upon receiving message $(\mathsf{update\text{-}ok}, s^{P'})$ from P', where $s^{P'}$ is $\mathsf{Sign}_{\mathsf{sk}_{P'}}(nid, \tilde{x}_a, \tilde{x}_b, \tilde{\sigma}, \textit{ver-num}^P + 1)$, proceed as follows:
   (a) Update local version of nanocontract with $\Gamma^P(id).\mathsf{nspace}(nid) := (\textit{blocked}, \tilde{\sigma}, \textit{ver-num}^P + 1, s^{P'})$, where $\textit{blocked}(\gamma.\mathsf{Alice}) := \tilde{x}_a$ and $\textit{blocked}(\gamma.\mathsf{Bob}) := \tilde{x}_b$.
   (b) Set $\Gamma^P(id).\mathsf{cash}(\gamma^P.\mathsf{Alice}) := \gamma^P.\mathsf{cash}(\gamma^P.\mathsf{Alice}) + x_a^P - \tilde{x}_a$ and $\Gamma^P(id).\mathsf{cash}(\gamma^P.\mathsf{Bob}) := \gamma^P.\mathsf{cash}(\gamma^P.\mathsf{Bob}) + x_b^P - \tilde{x}_b$ and return $\mathsf{updated}$.
8. Upon receiving message $(\mathsf{update\text{-}not\text{-}ok}, s^{P'})$ from P', where $s^{P'}$ is $\mathsf{Sign}_{\mathsf{sk}_{P'}}(nid, x_a^P, x_b^P, \sigma, \textit{ver-num}^P + 2)$, set $\Gamma^P(id).\mathsf{nspace}(nid).\mathsf{ver\text{-}num} := \textit{ver-num}^P + 2$ and $\Gamma^P(id).\mathsf{nspace}(nid).\mathsf{sign} := s^{P'}$ and send message $(\mathsf{update\text{-}not\text{-}ok}, s^P)$ with $s^P := \mathsf{Sign}_{\mathsf{sk}_P}(nid, x_a^P, x_b^P, \sigma^P, \textit{ver-num}^P + 2)$ to P'. Then return $\mathsf{notupdated}$.
9. If until time $\Upsilon + 1$ from the time you started $\texttt{UpdateStart}$ you did not receive any of the above messages from P', then call the subprocedure $\texttt{RegisterStart}(id, nid)$. Once the subprocedure returns $\mathsf{established}(\nu^*)$ and if $\nu^*.\mathsf{ver\text{-}num} > \textit{ver-num}^P$, then set:

$$\Gamma^P(id).\mathsf{cash}(\gamma^P.\mathsf{Alice}) := \gamma^P.\mathsf{cash}(\gamma^P.\mathsf{Alice}) + x_a^P - \tilde{x}_a \text{ and } \Gamma^P(id).\mathsf{cash}(\gamma^P.\mathsf{Bob}) := \gamma^P.\mathsf{cash}(\gamma^P.\mathsf{Bob}) + x_b^P - \tilde{x}_b.$$

Back to the procedure $\texttt{UpdateReply}(z, id, nid)$ run by P':

10. Upon receiving the message $(\mathsf{update\text{-}not\text{-}ok}, s^P)$ from P with $s^P := \mathsf{Sign}_{\mathsf{sk}_P}(nid, x_a^{P'}, x_b^{P'}, \sigma^{P'}, \textit{ver-num}^{P'} + 2)$ set $\Gamma^{P'}(id).\mathsf{nspace}(nid).\mathsf{ver\text{-}num} := \textit{ver-num}^{P'} + 2$ and $\Gamma^{P'}(id).\mathsf{nspace}(nid).\mathsf{sign} := s^P$.
11. Otherwise call the subprocedure $\texttt{RegisterStart}(id, nid)$. Once the subprocedure returns $\mathsf{established}(\nu^*)$, return $\mathsf{notupdated}$.

Fig. 19: Procedure for updating nanocontract (continued from the previous page).

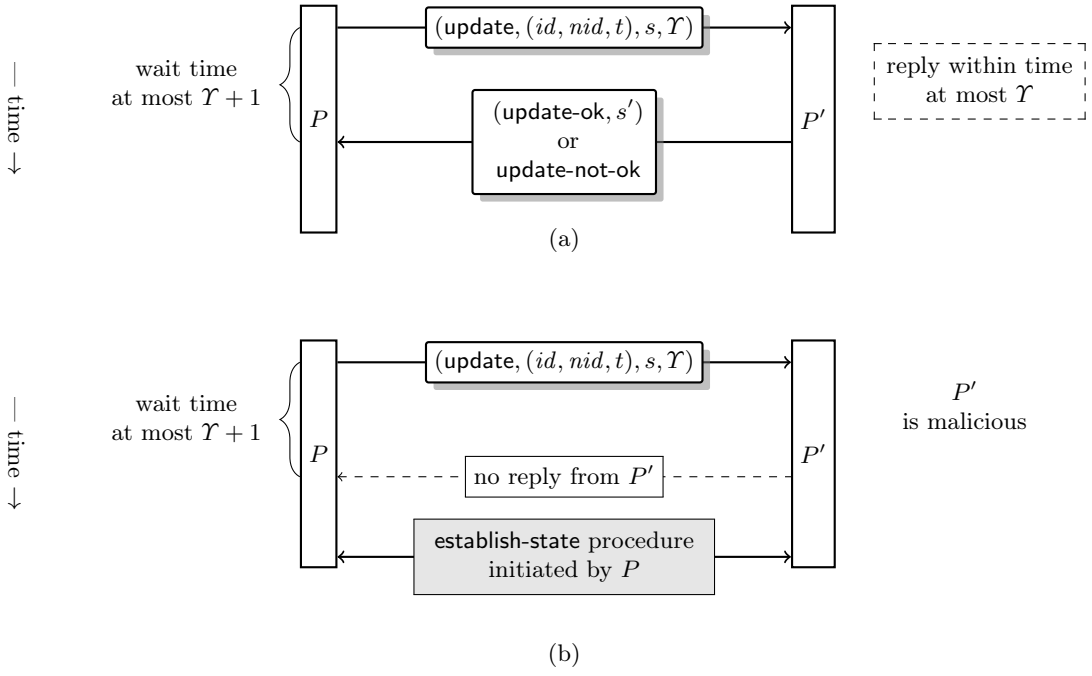


Fig. 20: Message flows in (a) a successful execution of the "nanocontract update" procedure, and (b) an *un*successful execution of the "nanocontract update" procedure.

The closing procedure works as follows. First, the parties need to inform the $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ about the current state of all the contracts (recall that at this point the state of some of the nanocontracts may still be off-chain). This is done as follows. The initiating party P runs the state registration subprotocol for every nanocontract in the channel, and at the same time it calls the function close in the contract $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$. Note that a dishonest P may try to cheat by not starting the state registration protocol for some of the existing nanocontracts. Therefore, $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ needs to give to P' the opportunity to execute the state registration subprotocol on the remaining nanocontracts. Hence, $\mathit{MSContract}^{\mathsf{C}}_{\gamma.\mathsf{id}}$ sends a closing message to P', and if P' can see that some of the nanocontracts have not been registered yet, run for these remaining nanocontracts the state registration procedure. The details of the closing procedure are presented on Fig. 21.

In the standard case when the parties agree then closing takes time $3 \cdot \Delta$, where $2 \cdot \Delta$ comes from registering the state, and Δ for the call to the function finalize-close(). In the pessimistic case state registration takes $3 \cdot \Delta$, so the total time is $4 \cdot \Delta$.

4.2.8 Using nanocontracts in practice. In the previous sections, we introduced protocols and contracts for implementing basic multistate channels in which nanocontracts can run. To complete our description, we need to give some guidelines how to use contracts in the multistate channels (they will also be used in Sect. 4.3 for creating the virtual channels).

Let C be some contract code and let π be contract parameters (with some fresh identifier $\pi.\mathsf{id}$). Suppose $\pi.\mathsf{Alice}$ and $\pi.\mathsf{Bob}$ created a basic multistate channel γ with $(\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}) = (\pi.\mathsf{Alice}, \pi.\mathsf{Bob})$ using protocol $\mathtt{MSChannels}^{\mathsf{C}}$ (see Sect. 4.2.3). Moreover assume that each $P \in \gamma.\mathsf{end\text{-}users}$ has at least $\pi.\mathsf{cash}(P)$ coins in her deposit in channel γ. Suppose that the parties agreed that in time in time τ they will start nanocontract C with parameters π in the channel γ. Then $\pi.\mathsf{Alice}$ and $\pi.\mathsf{Bob}$ start the `UpdateStart` and `UpdateWait` procedure (respectively) with parameters $(\gamma.\mathsf{id}, \mathit{nid}, \sigma, x_a, x_b, 0)$, where nid is chosen in such a way that it is unique (e.g. $\mathit{nid} = (\gamma.\mathsf{id}, \pi.\mathsf{id})$) and $(x_a, x_b) = (\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ and $\sigma := \mathsf{Init}(\tau, \pi)$. If as a result of the `UpdateWait` execution $\gamma.\mathsf{Bob}$ gets an output updated then he immediately runs the $\mathtt{UpdateReply}(\mathsf{ok}, \gamma.\mathsf{id}, \mathit{nid})$ procedure.

At the end either both $P \in \gamma.\mathsf{end\text{-}users}$ receive an output updated, in which case they conclude that the nanocontract with identifier nid has been created. Otherwise they both receive not-updated in which case they conclude that the contract C has not been created.

Now, suppose that the one of the parties $P \in \gamma.\mathsf{end\text{-}users}$ wants to execute a function *fun* with parameter z in the nanocontract with identifier nid in channel with identifier id. Then P can try to perform a "peaceful execution of the nanocontract". Let τ be the current time, and let $\sigma := \Gamma^P(\mathit{id}).\mathsf{nspace}(\mathit{nid}).\mathsf{storage}$. Party P simply contacts P' by sending to her a message $(\mathsf{optimistic}, \mathit{id}, \mathit{nid}, \mathit{fun}, z)$. Both parties compute $\mathsf{Eval}^{\mathsf{C}}(\sigma; P, 0, \mathit{fun}, z)$ locally, and update the nanocontract nid according to the output of this procedure. Only if this procedure fails, P will start the real execution of the nanocontract (to distinguish such execution from the "optimistic" one we will called it sometimes "forced execution"). For a detailed description of the optimistic execution procedure see Fig. 22.

The standard execution time of this procedure is 2 (since 2 rounds are needed for nanocontract update). The pessimistic execution time is $3 \cdot \Delta + 3$ (needed for the update) plus $4 \cdot \Delta$ for the forced execution. It is easy to see however, that this second value can be reduced to Δ, since the remaining $3 \cdot \Delta$ comes from the "state registration" that does not need to be performed if it was already performed in the channel update. Therefore the total pessimistic time of this procedure is $4 \cdot \Delta + 3$.

4.2.9 A discussion on parallelism. As already mentioned, we assume that the above procedures can be executed in parallel. There are some small natural limitation to this. In particular, we require that the parties do not establish channels that they "cannot afford", and before a channel creation procedure starts they check if they don't execute another channel creation procedure that uses the same coins. The same restrictions apply to the nanocontract update.

4.2.10 Some optimizations and extensions. One obvious drawback of the above construction is that it requires the parties to store forever all the ν tuples that every appeared for a given multistate channel, just in case the other party will post some old version of ν on the ledger. Also, in the closing procedure the parties need to send every ν to *MSContract*. Fortunately, our protocol can be easily modified to get rid of this problem. This can be done by allowing the parties to "clean" $\gamma.\mathsf{nspace}$ in the following way. Every channel γ will contain a special list of inactive (expired) nanocontracts, from time to time the parties will run an update procedure that will update this list. This can be optimized further, by having special field in γ that invalidates all nanocontracts satisfying certain condition, e.g. those whose identifier starts with certain string, or those that are were created before some time (in this case *nid*'s needs to contain also the timestamp that indicates when they were created).[18] In the simplest case (when there is no conflict between the users) they can simply make an update that removes all nanocontract (or, in other words, "converts the multistate channel into a payment channel).

Another limitation of our construction is that we do not allow the users to add coins into a nanocontract during the execution (cf. "0" in the argument of Eval in Sect. 4.2.5). We note that it is possible to allow it, however, for this we would need to make the contract *MSContract* "aware" of the current value of $\gamma.\mathsf{cash}$. For this, we would need to execute the state establishment procedure for *every* nanocontract in the channel, before the execution starts. This extension is discussed in Sect. 4.4.

4.3 Virtual payment channels

As already highlighted in the introduction the main contribution of this paper is a method for "channel virtualization". In this paper we focus here on virtual *payment* channels, but our ideas can be extended to virtual state channels (see Sect. 4.4). In virtual payment channels, we consider a setting with 3 parties called Alice, Ingrid, and Bob (where "Ingrid" connotes with the "intermediary").

The virtual channel protocol that we denote VirtualChannels is built "on top" of a basic state channel protocol $\mathtt{MSChannels}^{C}$, where $C = \mathit{VPC}$ is a contract that we call a *virtual payment contract*, which is presented on Fig. 24, and that is explained in detail later in this section . Suppose the parties already created basic state channels (Alice $\overset{\gamma_a}{\Leftrightarrow}$ Ingrid) and (Ingrid $\overset{\gamma_b}{\Leftrightarrow}$ Bob) using the $\mathtt{MSChannels}^{\mathit{VPC}}$ protocol. Our main idea is to build a virtual channel over these two basic nanocontract channels, where the intermediate party Ingrid acts as "payment hub" between Alice and Bob. In contrast to current payment networks such as the Lightning network, in our construction the intermediate Ingrid is not involved in all the individual payments that may be carried out between Alice and Bob. Instead, Ingrid is solely needed for the initial creation of the virtual payment channel and for its closing (which may involve handling disputes between Alice and Bob). Hence, conceptually Ingrid's role in our channel virtualization approach is similar to the role that the blockchain takes for the basic channels.

Similar to the *PaymentContract* from the basic payment channels, the main goal of the *VPC* is to guarantee a distribution of the money in the virtual channel according to its latest state. An

[18] Another option to avoid the need for remembering old nanocontracts is to simply "reuse" their identifiers (once a nanocontract became inactive, use its identifier for a new nanocontract).

important difference, however, is that in our protocols whenever there is a dispute between Alice and Bob, the parties first try to resolve this dispute through Ingrid, i.e., without touching the blockchain. If this first stage fails, then the execution of the VPCs gets escalated to the blockchain. The advantage of this two stage approach is that in the case when, for instance, Alice and Ingrid are honest, they do not need to touch the blockchain when they want to resolve a dispute between Alice and Bob, and hence conflict resolution can be instantaneous.

In the following subsections we will describe in detail how virtual payment channels are realized at a technical level. We start in Section 4.3.1 by introducing some syntax that is needed for the virtual payment channels. Then, in Sect 4.3.2 we given an a overview of the scheme, and in Sections 4.3.3, 4.3.4, and 4.3.5, we describe the sub-protocols for the channel creation, update, and closing (respectively).

4.3.1 Syntax for virtual payment channels. We start with describing the syntax that is needed for the exposition of our protocol; most of the notation is borrowed from the syntax used by the basic payment channels. A *virtual payment channel* γ *over a set of players* $\mathcal{P}$ is an attribute tuple of the form:

$$(\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Ingrid}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{subchan}, \gamma.\mathsf{validity}), \tag{6}$$

where $\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}$ and $\gamma.\mathsf{cash}$ are as in Eq. (4) in Section 4.1.1 (in particular: we assume that $\gamma.\mathsf{id}$ is freshly chosen), $\gamma.\mathsf{Ingrid} \in \mathcal{P}$ is a party that we will sometimes call the intermediary. We also have that $\gamma.\mathsf{subchan}$ is a function from $\gamma.\mathsf{end\text{-}users}$ to $\{0,1\}^*$, where $\gamma.\mathsf{end\text{-}users} = \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$. Finally $\gamma.\mathsf{validity} \in \mathsf{time}$ denotes the *channel validity*, i.e., the time until which the virtual payment channel stays open (we discuss the need for such a parameter below). Moreover $\gamma.\mathsf{end\text{-}users}$ and $\gamma.\mathsf{total\text{-}cash}$ are defined as for the case of basic channels.

For $P \in \gamma.\mathsf{end\text{-}users}$ the value $\gamma.\mathsf{subchan}(P)$ will be called the *identifier of* P*'s subchannel*, and it will be used to indicate the corresponding identifier of the basic multistate channel (constructed using $\mathtt{MSChannels}^{\mathit{VPC}}$) that is used to construct γ. That is, the values $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ and $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$ are pointers to basic channels $(\gamma.\mathsf{Alice} \overset{\gamma_a}{\Leftrightarrow} \gamma.\mathsf{Ingrid})$ and $(\gamma.\mathsf{Ingrid} \overset{\gamma_b}{\Leftrightarrow} \gamma.\mathsf{Bob})$, respectively. To distinguish virtual channels from basic channels we will denote them by $\gamma.\mathsf{Alice} \overset{\gamma}{\leftrightarrow} \gamma.\mathsf{Bob}$, i.e., we use a single-lined arrow.

The purpose of having $\gamma.\mathsf{validity}$ is as follows. In our construction every virtual channel will have a corresponding nanocontract in the basic subchannels γ_a and γ_b. Recall from Sect. 4.1 that the basic channels cannot be closed as long as nanocontracts are active. Therefore, a malicious Alice or Bob could block the channels γ_a and γ_b forever, by not closing the virtual channel γ. We prevent this by adding an automatic closure time $\gamma.\mathsf{validity}$ when the corresponding nanocontracts will be closed.

Observe that, since in this section we are only interested in virtual *payment* channels, Eq. (6) does not have a $\mathsf{storage}$ attribute. We discuss the possibility of creating virtual *nanocontract* channels in Section 4.4. Another restriction that we have is that the channels are just of length 2 (i.e. there is *one* intermediary). An extension of our idea to longer channels is also discussed in Section 4.4.

4.3.2 An overview of the construction. Technically, the system that we construct will be an extension of the basic multistate channel system from Sect. 4.2, with contract code C being equal to VPC (defined below). In other words, we assume that all the procedures for handling the basic channels are handled by the $\mathtt{MSChannels}^{\mathit{VPC}}$ protocol, and the only thing that is needed is to specify the procedures for the virtual channels. A *channel space* is defined in the same way as the the basic

Channel closing by one of the parties

To close a basic multistate channel with identifier id, a party P (that is one of the end-users of this channel) proceeds as follows (let $P' := \gamma.\mathsf{other\text{-}party}(P)$):

Procedure $\texttt{CloseStart}(id)$ for party P

1. Do the following in parallel:
   - for each nid such that $\Gamma^P(id).\mathsf{nspace}(nid) \neq \bot$ and the state registration procedure has not been run for this nanocontract before, call the $\texttt{RegisterStart}(id, nid)$, and
   - call the function $\mathsf{close}()$ in $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$.
2. Upon receiving a message $\mathsf{closing}$ from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ wait at most 3Δ rounds.

Procedure $\texttt{CloseReact}(id)$ for party P':

3. Upon receiving a message $\mathsf{closing}$ from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ do the following:
   - for each nid such that $\Gamma^{P'}(id).\mathsf{nspace}(nid) \neq \bot$ and the state registration procedure has not been run for this nanocontract before, call the $\texttt{RegisterStart}(id, nid)$.
4. Then wait.

Back to procedure $\texttt{CloseStart}(id)$ for party P

After time 3Δ has passed from Step (2), or when all nanocontracts in $\Gamma^P(id)$ have been registered, then call the function $\mathsf{finalize\text{-}close}()$ in $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$.

Rest of the procedures $\texttt{CloseStart}(id)$ and $\texttt{CloseReact}(id)$ for $T = P, P'$ (respectively):

5. Upon receiving a message closed from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ erase γ from Γ^T and return closed.
6. Upon receiving a message $\mathsf{not\text{-}closed}$ from $\mathit{MSContract}^{\mathcal{C}}_{\gamma.\mathsf{id}}$ return $\mathsf{not\text{-}closed}$.

(a)

Function $\mathsf{close}()$ **called by** $P \in \mathsf{end\text{-}users}$

Store $(P, \mathsf{close}, \mathsf{current\text{-}time})$ and send $\mathsf{closing}$ to both parties in $\gamma.\mathsf{end\text{-}users}$. Then go idle.

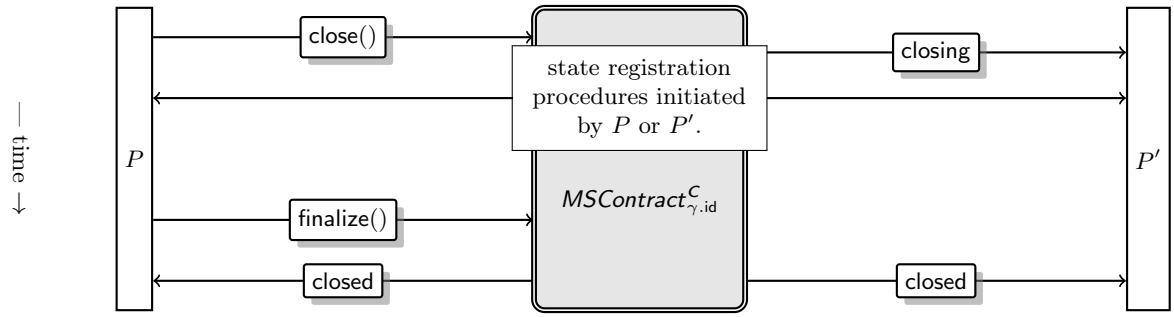
Function $\mathsf{finalize\text{-}close}()$ **called by** $P \in \mathsf{end\text{-}users}$

If your storage contains a value $(\widehat{P}, \mathsf{close}, \widehat{\tau})$ such that $\widehat{P} = P$ and $\mathsf{current\text{-}time} > 3\Delta + \widehat{\tau}$, then do the following:

- if there exists nid such that $\gamma.\mathsf{nspace}(nid).\mathsf{total\text{-}blocked} > 0$, then send a message $\mathsf{not\text{-}closed}$ to both users in $\gamma.\mathsf{end\text{-}users}$ and go idle,
- otherwise send to each $T \in \gamma.\mathsf{end\text{-}users}$ a message closed together with $\gamma.\mathsf{cash}(T)$ coins and terminate.

(b)

Fig. 21: (a) Procedure for closing the basic multistate channel, and (b) the corresponding functions in the contract.


time →
P
close()
state registration procedures initiated by P or P′.
closing
P′
finalize()
MSContract$^{C}_{\gamma.\mathsf{id}}$
closed
closed

Optimistic nanocontract execution

To *optimistically* execute a nanocontract with identifier nid in the basic multistate channel γ with identifier id, a party P (that is one of the end-users of this channel) proceeds as follows (let $P' := \gamma.\mathsf{other\text{-}party}(P)$):

Procedure `OptimisticExecStart`$(id, nid, \mathit{fun}, z)$ for party P

1. If you have already executed the `RegisterStart` or `RegisterReact` procedure with parameters (id, nid) then go to Step 6.
2. Otherwise send a message $(\mathsf{optimistic}, id, nid, \mathit{fun}, z)$ to P'.
3. Let τ be the current time, and let $\sigma := \Gamma^P(id).\mathsf{nspace}(nid).\mathsf{storage}$. Compute

$$(\widehat{\sigma}; m_a, y_a; m_b, y_b) := \mathsf{Eval}^{\mathsf{C}}(\sigma; P, 0, \tau; \mathit{fun}, z).$$

   Execute the `UpdateStart` procedure with parameters

$$\mathit{params} = (\gamma.\mathsf{id}, nid, \widehat{\sigma}, x_a + y_a, x_b + y_b, 0),$$

   where nid is chosen in such a way that it is unique (e.g. $nid = (\gamma.\mathsf{id}, \pi.\mathsf{id})$) and $(x_a, x_b) = (\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$, and wait 1 round.

Procedure `OptimisticExecReact` for party P'

4. Upon receiving a message $(\mathsf{optimistic}, id, nid, \mathit{fun}, z)$ (that P sent in Step 2):
   If in the previous round you started `OptimisticExecStart` with the same id and nid and if $P' = \gamma.\mathsf{Bob}$ then stop (This technical step is needed to handle the unlikely case when both parties simultaneously decided to execute the same nanocontract. To "break the symmetry" of the channel we assume that in this case Bob "behaves nicely" and gives Alice the priority.)
   If you have already executed the `RegisterStart` or `RegisterReact` procedure with parameters (id, nid) then do nothing.
   Otherwise let τ be the current time, $\sigma' = \Gamma^{P'}(id).\mathsf{nspace}(nid).\mathsf{storage}$ and compute

$$(\widehat{\sigma}'; m'_a, y'_a; m'_b, y'_b) := \mathsf{Eval}^{\mathsf{C}}(\sigma'; P, 0, \tau - 1; \mathit{fun}, z).$$

   If $\widehat{\sigma}' = \mathsf{ignore}$ then do nothing. Otherwise execute the `UpdateWait` procedure with parameters

$$\mathit{params}' = (\gamma.\mathsf{id}, nid, \widehat{\sigma}', x_a + y'_a, x_b + y'_b, 0),$$

   (if P and P' are honest then $\sigma' = \sigma$ and hence $(\widehat{\sigma}'; m'_a, y'_a; m'_b, y'_b) = (\widehat{\sigma}; m_a, y_a; m_b, y_b)$ and $\mathit{params}' = \mathit{params}$). If, as a result of this execution, you get as output $\mathsf{update\text{-}requested}$ then immediately run the `UpdateReply`$(\mathsf{ok}, \gamma.\mathsf{id}, nid)$ procedure and go to Step 7.

Back to procedure `OptimisticExecStart`$(id, nid, \mathit{fun}, z)$ for party P

5. If `UpdateStart` returns $\mathsf{updated}$ then go to Step 7.
6. Otherwise run `ExecStart`$(id, nid, \mathsf{fun}, z)$.

Rests of procedures `OptimisticExecStart`$(id, nid, \mathit{fun}, z)$ and `OptimisticExecReact` for $T = P$ and $T = P'$ respectively

7. Change $\gamma' \in \Gamma^{P'}(id).\mathsf{nspace}$ by adding y_a and y_b coins to the deposits of $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ (resp) and change $\nu \in \gamma'.\mathsf{nspace}$ as follows:
   - $\nu.\mathsf{storage} := \widehat{\sigma}'$,
   - $\nu.\mathsf{cash}(\gamma.\mathsf{Alice}) := \nu.\mathsf{cash}(\gamma.\mathsf{Alice}) - y_a$,
   - $\nu.\mathsf{cash}(\gamma.\mathsf{Bob}) := \nu.\mathsf{cash}(\gamma.\mathsf{Bob}) - y_b$,
8. Return $(\mathsf{executed}, id, nid, m_a)$ (if $T = \gamma.\mathsf{Alice}$) or $(\mathsf{executed}(id, nid, m_b)$ (if $T = \gamma.\mathsf{Bob}$).

Fig. 22: Optimistic nanocontract execution

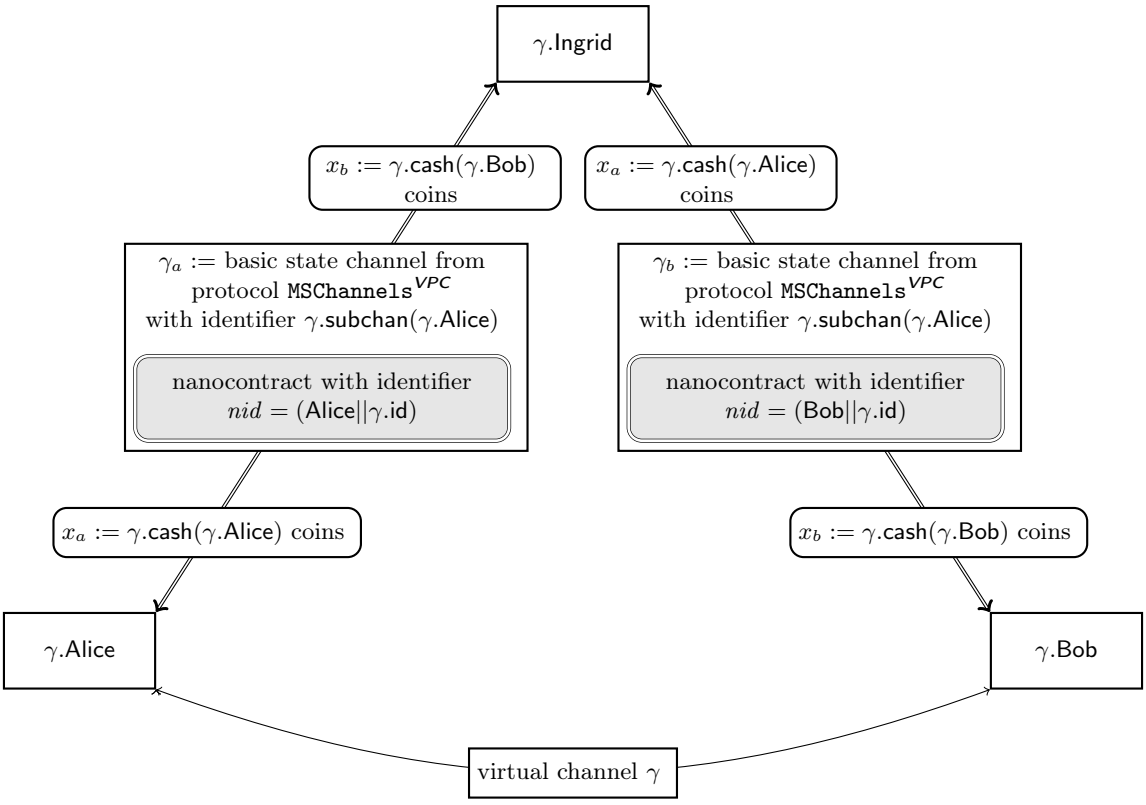


Fig. 23: A virtual channel built "on top" of two basic channels.

payment channel space (see Sect. 4.1.1). It will now contain both the virtual channels and the basic channels.

To build a virtual channel γ "on top" of two basic channels γ_a and γ_b, we create two nanocontracts ν_a and ν_b with identifiers $(\mathsf{Alice}, \gamma.\mathsf{id})$ and $(\mathsf{Alice}, \gamma.\mathsf{id})$ in γ_a and γ_b, respectively. These nanocontracts are responsible for handling the channel γ and work as follows. Initially, party $\gamma.\mathsf{Alice}$ will have $x_a = \gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ coins blocked in ν_a, while $\gamma.\mathsf{Ingrid}$ will have $x_b = \gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ coins blocked in it. Symmetrically, $\gamma.\mathsf{Bob}$ will initially have x_b coins blocked in ν_b, and $\gamma.\mathsf{Ingrid}$ will have x_a coins blocked in it.

Similarly to what was done in the previous sections, the end-users of a virtual payment channel γ, will internally extend γ by adding two attributes to it: $\gamma.\mathsf{ver\text{-}num} \in \mathbb{N}$ and $\gamma.\mathsf{sign} \in \{0,1\}^*$. We say that such an extended γ is a *version of γ's state signed by $P \in \gamma.\mathsf{end\text{-}users}$* if $\gamma.\mathsf{ver\text{-}num} \geq 1$ and $\gamma.\mathsf{sign}$ is a signature of P on $(\gamma.\mathsf{id}, \gamma.\mathsf{cash}, \gamma.\mathsf{ver\text{-}num})$. When $\gamma.\mathsf{ver\text{-}num} = 0$ the attribute $\gamma.\mathsf{sign}$ will be undefined, but for consistency also in this case we say that $(\gamma.\mathsf{id}, \gamma.\mathsf{cash}, \gamma.\mathsf{ver\text{-}num})$ is a version of γ's state signed by any $P \in \gamma.\mathsf{end\text{-}users}$. Recall that the idea that we had for the basic payment channels was: in every update the end-users of γ send to each other a signed new version of γ with $\gamma.\mathsf{ver\text{-}num}$ incremented by 1, In case of a dispute between them, the contract *StateChannelContract* that parametrized the system could decide what is the latest version of γ by looking at the $\gamma.\mathsf{ver\text{-}num}$ numbers and checking the signatures.

The main challenge in designing the *virtual* payment channels is that we want $\gamma.\mathsf{Ingrid}$ to be the entity that resolves conflicts between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$, while we do not want to make any trust assumptions about her. The important thing from Alice's point of view is that she has to be sure that when she sends to $\gamma.\mathsf{Ingrid}$ her latest version of γ, then $\gamma.\mathsf{Ingrid}$ will transfer $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ coins from ν_a to $\gamma.\mathsf{Alice}$'s deposit in the basic channel γ_a. Of course, Bob needs to have a symmetric guarantee, and finally, also $\gamma.\mathsf{Ingrid}$ needs to be sure that she can get $\gamma.\mathsf{cash}(\gamma.\mathsf{Bob})$ coins from nanocontract ν_a, and $\gamma.\mathsf{cash}(\gamma.\mathsf{Alice})$ coins from nanocontract ν_b. The exact mechanics of this is described in Sect. 4.3.5 which contains the protocol for closing a channel. First, however, we present the procedures for creating and updating the virtual channels.

Virtual contract contract *VPC*

Let γ be a virtual channel. The bilateral contract *VPC* is initialized with an attribute tuple π such that either

- $(\pi.\mathsf{Alice}, \pi.\mathsf{Bob}) := (\gamma.\mathsf{Ingrid}, \gamma.\mathsf{Bob})$,

and $(\pi.\mathsf{cash}(\pi.\mathsf{Alice}), \pi.\mathsf{cash}(\pi.\mathsf{Bob})) := (\gamma.\mathsf{cash}(\gamma.\mathsf{Alice}), \gamma.\mathsf{cash}(\gamma.\mathsf{Bob}))$. Moreover π has one additional attribute "virtual-channel" defined as $\pi.\mathsf{virtual\text{-}channel} := \gamma$. The contract consist of the following functions.

- $\mathsf{close}(\gamma^P)$ and
- $\mathsf{finalize}()$.

Both of them are described on Fig. 30.

Fig. 24: The Virtual Payment Contract. This contract interacts with the "channel closing" procedure, see Sect. 4.3.5, page 48.

4.3.3 Creating a virtual payment channel. Before describing how the nanocontracts ν_a and ν_b take care of fair money distribution, let us show how a virtual payment channel γ is created. Let $x_a, x_b, \gamma_a, \gamma_b, nid_a$, and nid_b be defined as in the beginning of Sect. 4.3.2. Initially $\gamma.\mathsf{Alice}$ puts x_a

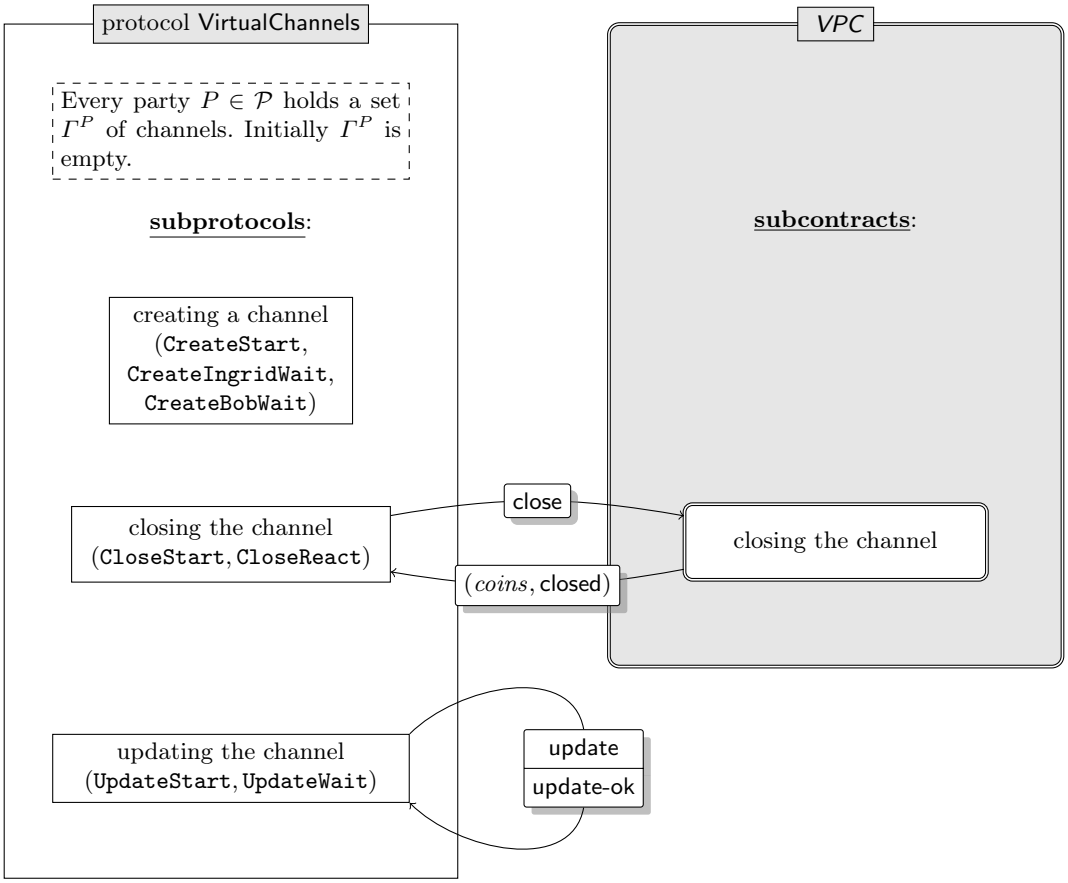


Fig. 25: The general structure of the scheme for virtual payment channels, including the messages that are sent between the parties and the contract, and the names of the procedures.

and γ.Bob puts x_b coins into the virtual channel. Hence, nobody loses money if the channel closes in its initial state. The channel creation sub-protocol is presented on Fig. 26, and the graphical representation of the procedure calls is depicted on Fig. 28.

The protocol for creating a virtual channel is run between the parties γ.Alice, γ.Bob and γ.Ingrid and starts with the three procedures `CreateStart` (executed by γ.Alice), `CreateIngridWait` (executed by γ.Ingrid), and `CreateBobWait` (executed by γ.Bob) depicted on Fig. 26. We assume that all the parties start execution of their procedures in the same round (τ_{init}). Initially (see Step 0), each of them "internally" blocks the corresponding coins in γ_a and γ_b, so they cannot be used for any other purpose until γ is closed, or it becomes clear that γ will not be created.

Then, in Step 1 , γ.Alice proposes to γ.Ingrid to update a channel γ_a by creating a nanocontract ν_a and moving γ.Alice's x_a coins and γ.Ingrid's x_b coins to the nanocontract ν_a. By "proposing" we mean that γ.Alice starts the `UpdateWait` procedure from $\texttt{MSChannels}^{\mathcal{VPC}}$ with parameters $(\gamma_a.id, nid, \mathsf{Init}(\gamma), x_a, x_b, 3 \cdot \Delta + 1)$. Concretely, this update should result in creating a new nanocontract (with identifier nid_a) that handles γ in γ_a. The initial storage of this nanocontract is $\mathsf{Init}(\gamma)$, the values x_a and x_b are the funds that shall be blocked in the nanocontract and $3 \cdot \Delta + 1$ is the amount of time that γ.Alice gives to γ.Ingrid in addition to the standard execution time needed for the update subprotocol to finish in the pessimistic case.

Party γ.Ingrid starts her execution of the channel creation process by waiting for a signal that γ.Alice proposed an update. Technically, this is done by starting the `UpdateWait` procedure and waiting to obtain update-requested (see Step 2). Once such a signal is obtained (in round $\tau_{\mathsf{init}} + 1$, see Step (3a)) then γ.Ingrid does not confirm to γ.Alice the update of γ_a immediately. This is because γ.Ingrid cannot be sure if γ.Bob will also agree for the corresponding update of γ_b. Notice that if ν_a is created, and ν_b is *not* created, then γ.Ingrid is "exposed" in the following way: she has to honor γ.Bob's obligations with respect to γ.Alice (i.e. pay to γ.Alice all the money that γ.Bob has to pay in channel γ), but she *cannot* get this money back from γ.Bob (since officially channel γ_b is not "aware" of the fact that γ has been created). Therefore before confirming the update to γ.Alice, γ.Ingrid asks γ.Bob to update the channel γ_b by creating the nanocontract ν_b and moving γ.Ingrid's x_a coins and γ.Bob's x_b coins to a ν_b. This is done by starting the `UpdateStart` procedure with parameters $params_b$ (see Eq. (8), Step (3a)). If on the other hand γ.Ingrid does not obtain update-requested from her call to `UpdateWait` then she concludes that γ.Alice does not intend to create a channel γ, and unblocks the money that she blocked in Step 0 and stops (this happens in Step (3b)).

Finally, let us take a look how party γ.Bob starts the channel creation process. He calls the procedure `UpdateWait`($params_b$) and waits to obtain update-requested (see Step 4). If such a signal is obtained within round $\tau_{\mathsf{init}} + 2$ (see Step (5a)) then γ.Bob confirms the update by calling `UpdateOK` procedure with input ok, which notifies γ.Ingrid that the update of channel γ_a was completed, and contains the nanocontract ν_a. Otherwise he concludes that one of the other parties does not intend to create a channel γ, unblocks the money that he blocked in Step 0 and stops (this happens in Step (5b)).

Only once γ.Ingrid receives information from the `UpdateWait` procedure that γ.Bob confirmed the update, she sends back her update confirmation to γ.Alice (this happens in Step (6a). If γ.Ingrid receives information from the `UpdateWait` that γ.Bob did not confirm the update then she informs γ.Alice that channel γ cannot be created, by starting an `UpdateReply` procedure with input not-ok (see Step (6b). Recall that execution of `UpdateWait` can take pessimistic time $3 \cdot \Delta$ (since it may involve the "state registration" procedure if the parties do not cooperate). Therefore the pessimistic execution time of this part is $1 + 3 \cdot \Delta$. This is exactly the reason why γ.Alice waits additionally time Υ.

Finally, γ.Alice receives an output from the `UpdateStart` procedure. If it is updated then she concludes that the channel has been created, and sends a final confirmation message channel-ok to

γ.Bob (see Step (7a)). Otherwise she concludes that one of the other parties does not intend to create a channel γ, unblocks the money that she blocked in Step 0 and stops. (this happens in Step (7b)). Once γ.Bob receives message channel-ok he also concludes that the channel has been created (see Step 8).

Observe that in our procedure the order of signing the channel updates is γ.Alice $\to$ γ.Ingrid $\to$ γ.Bob $\to$ γ.Ingrid $\to$ γ.Alice (see Fig. 27). Therefore the only party that potentially can be "exposed" is γ.Alice, in the sense that only γ.Alice signs an update to a channel (γ_a) before receiving a corresponding signature from the other party. The reason why it is not a problem is that the only way in which γ.Ingrid could use the γ.Alice's signature on channel update is to post it on the blockchain together with her signature s_b (otherwise the contract would not accept it). In this way, however, she reveals s_b, so it can as well count as "sending s_b to γ.Alice". Sending the confirmation to γ.Bob in the last step is needed to guarantee that, in case both γ.Alice and γ.Bob are honest, they will have an identical view on the fact whether the channels was created. The total pessimistic time of the execution of this procedure is $2 \cdot (1 + 3 \cdot \Delta) + 1 = 6 \cdot \Delta + 3$. The standard time is 5 rounds (since we just need to count the rounds in which the messages are sent between the parties).

4.3.4 Updating a channel. Let *id* be an identifier of some virtual payment channel γ. Updating the state of γ (i.e. the value of cash) is implemented in the same way as for the basic payment channels (see Sect. 4.1.4 with a detailed description, including the message flow on Fig. 8, page 18). The optimistic execution of the update takes 2 rounds. The pessimistic time is only bounded by γ.validity.

4.3.5 Closing a channel. The tuples that the parties γ.end-users store after an update have practical meaning only if they can be sure that Ingrid will "honor them". Technically, as already described in Sect. 4.3.2, this will be guaranteed by the nanocontracts ν_a and ν_b in the channels γ_a and γ_b (respectively). Let us now informally discuss how to guarantee that every party is treated fairly (the subprotocol for channel closing is depicted on Fig. 29, the corresponding contract functions are presented on Fig. 30, and the message flows are on Figures 31 and 32). Assume that a virtual channel γ has already been created, and suppose time γ.validity comes and the channel should be closed. For a moment assume that γ.Ingrid is a trusted party. Consider the following procedure in which γ.Ingrid determines what is the latest state version $cash : \gamma.\text{end-users} \to \mathbb{R}_{\geq 0}$ of γ (below let $id = \gamma.\text{id}$).

1. Each $P \in \gamma$.end-users sends her tuple $\gamma_P := \Gamma^P(id)$ to γ.Ingrid,
2. Upon receiving γ_P from P in Step 1, party γ.Ingrid checks if γ_P is correctly signed by γ.other-party(P) and accepts it only if the signature is correct. Then, consider the following cases:
   (a) If both $\gamma_{\gamma.\text{Alice}}$ and $\gamma_{\gamma.\text{Bob}}$ are correctly signed, then γ.Ingrid checks which of these messages has a higher version number. Let γ^* be this message (choose γ^* arbitrarily if the version numbers are equal). Ingrid assumes that the latest version *cash* of the channel with identifier $\gamma.id$ is γ^*.cash.
   (b) only one correctly signed γ_P was received by γ.Ingrid — in this case γ.Ingrid assumes that $cash := \gamma_P$.cash.
   (c) no correctly signed γ_P was received by γ.Ingrid — in this case she assumes that $cash := \gamma_{\text{init}}$.cash, where γ_{init} is the initial version of the channel (when it was created).

Subprotocol for creating a virtual payment channel

To create the virtual payment channel γ the parties proceed as follows (we assume that all parties P start in the same round τ_{init}, and we let $x_a, x_b, \gamma_a, \gamma_b, nid_a$, and nid_b be as defined at the beginning of Sect. 4.3.2).

Procedures $\texttt{CreateStart}(\gamma)$, $\texttt{CreateIngridWait}(\gamma)$ and $\texttt{CreateBobWait}(\gamma)$ run by $\gamma.\mathsf{Alice}$, $\gamma.\mathsf{Ingrid}$, and $\gamma.\mathsf{Bob}$ (respectively)

0. Party $\gamma.\mathsf{Alice}$ removes x_a coins from her deposit in channel $\gamma_a \in \Gamma^{\gamma.\mathsf{Alice}}$.
   Party $\gamma.\mathsf{Ingrid}$ removes x_b and x_a coins from her deposits in channels with identifiers γ_a and γ_b (respectively) in $\Gamma^{\gamma.\mathsf{Ingrid}}$.
   Party $\gamma.\mathsf{Bob}$ removes x_b coins from his deposit in channel $\gamma_b \in \Gamma^{\gamma.\mathsf{Bob}}$.

Procedure $\texttt{CreateStart}(\gamma)$ run by $\gamma.\mathsf{Alice}$:

1. Call procedure $\texttt{MSChannels}^{\mathcal{VPC}}.\texttt{UpdateStart}(params_a)$, where the parameters are defined as follows:
$$params_a := (\gamma_a.\mathsf{id}, nid_a, \mathsf{Init}(\gamma), x_a, x_b, 3 \cdot \Delta + 1) \tag{7}$$
   and wait.

Procedure $\texttt{CreateIngridWait}(\gamma)$ run by $\gamma.\mathsf{Ingrid}$:

2. Call $\texttt{MSChannels}^{\mathcal{VPC}}.\texttt{UpdateWait}(params_a)$ and wait until at most round $\tau_{\mathsf{init}} + 1$ to obtain the return $\mathsf{update\text{-}requested}$.
3. Distinguish two cases:
   (a) $\mathsf{update\text{-}requested}$ *returned:* call $\texttt{MSChannels}^{\mathcal{VPC}}.\texttt{UpdateStart}(params_b)$ of the basic state channel protocol, where the parameters are defined as follows:
$$params_b := (\gamma_b.\mathsf{id}, nid_b, \mathsf{Init}(\gamma), x_a, x_b, 0). \tag{8}$$
   and wait.
   (b) *Otherwise:* Unblock the money that you removed in Step 0 in your deposits in the channels in $\Gamma^{\gamma.\mathsf{Ingrid}}$ and stop.

Procedure $\texttt{CreateBobWait}(\gamma)$ run by $\gamma.\mathsf{Bob}$:

4. Call $\texttt{MSChannels}^{\mathcal{VPC}}.\texttt{UpdateWait}(params_b)$. Wait until at most round $\tau_{\mathsf{init}} + 2$ to obtain the return $\mathsf{update\text{-}requested}$.
5. Distinguish two cases:
   (a) $\mathsf{update\text{-}requested}$ *returned:* call $\texttt{MSChannels}^{\mathcal{VPC}}.\texttt{UpdateReply}(\mathsf{ok}, \gamma_b.\mathsf{id}, nid_b)$ and wait until the "channel closing subprotocol" is started at time $\gamma.\mathsf{validity}$ (see Sect. 4.3.5).
   (b) *Otherwise:* unblock the money that you removed in Step 0 in your deposits in channel γ_b in $\Gamma^{\gamma.\mathsf{Bob}}$ and stop.

Back to procedure $\texttt{CreateIngridWait}$ run by $\gamma.\mathsf{Ingrid}$:

6. Wait at most until round $\tau_{\mathsf{init}} + 3 \cdot \Delta + 1$ for an output of $\texttt{MSChannels}^{\mathcal{VPC}}.\texttt{UpdateStart}(params_b)$, and proceed as follows:
   (a) $\mathsf{updated}$ *returned:* Add γ to $\Gamma^{\gamma.\mathsf{Ingrid}}$, call $\texttt{MSChannels}^{\mathcal{VPC}}.\texttt{UpdateReply}(\mathsf{ok}, \gamma_a.\mathsf{id}, nid_a)$, and wait until the "channel closing subprotocol" is started at time $\gamma.\mathsf{validity}$.
   (b) $\mathsf{not\text{-}updated}$ *returned:* $\gamma.\mathsf{Ingrid}$ calls $\texttt{MSChannels}^{\mathcal{VPC}}.\texttt{UpdateReply}(\mathsf{not\text{-}ok}, \gamma_a.\mathsf{id}, nid_a)$. Then, it unblocks x_b coins in $\gamma_a \in \Gamma^{\gamma.\mathsf{Ingrid}}$ and stops.

Subprotocol for creating a virtual payment channel (continued on the next page).

Subprotocol for creating a virtual payment channel (continued)

Back to procedure `CreateStart` run by γ.Alice:

7. Wait at most until round $\tau_{\mathsf{init}} + 6 \cdot \Delta + 2$ for $\texttt{MSChannels}^{\mathit{VPC}}.\texttt{UpdateStart}(params_a)$, and distinguish the following cases:
   (a) updated *returned:* Send a message $(\mathsf{channel\text{-}ok}, \gamma.\mathsf{id})$ to γ.Bob. Then, add γ to $\Gamma^{\gamma.\mathsf{Alice}}$ and wait until the "channel closing subprotocol" is started at time γ.validity.
   (b) not-updated *returned:* γ.Alice unblocks the money that it previously locked in γ_a and updates $\Gamma^{\gamma.\mathsf{Alice}}$ and stops.

Back to procedure `CreateStart` run by γ.Bob:

8. If until round $\tau_{\mathsf{init}} + 6 \cdot \Delta + 3$ you receive a message $(\mathsf{channel\text{-}ok}, \gamma.\mathsf{id})$ from γ.Alice, then add γ to $\Gamma^{\gamma.\mathsf{Bob}}$.
9. Wait for the "channel closing subprotocol" at time γ.validity.

Fig. 26: Subprotocol for creating a virtual payment channel (continued from the previous page).

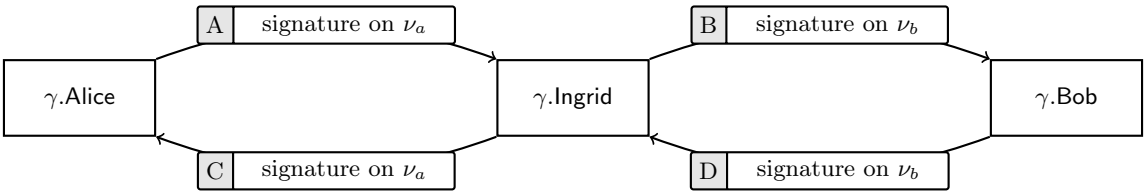


Fig. 27: The order of signing the channel updates in the procedure for virtual channel creation.

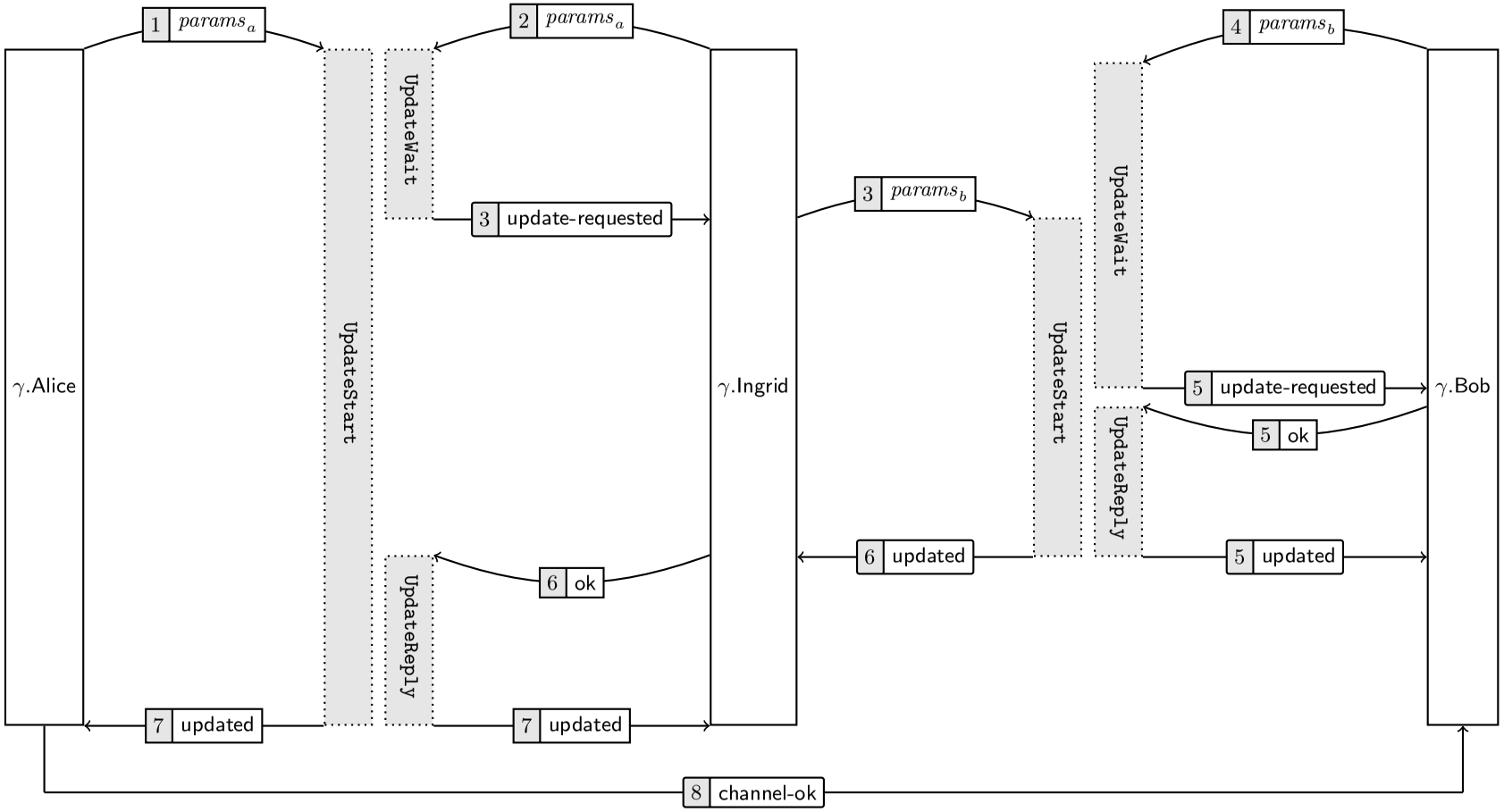


Fig. 28: Graphical representation of the procedure calls (to the $\texttt{MSChannels}^{VPC}$ procedures) in a successful execution of a sub-protocol for creating a virtual payment channel. The numbers in gray boxes correspond to steps on Fig. 26. The arc arrows correspond to parameters passed as inputs to the procedures, and the straight arrows correspond to the outputs of the procedures.

Once γ.Ingrid establishes the value of *cash* she transfers the coins in channels γ_a and γ_b in the following way (let $(x_a, x_b) := (cash(\gamma.\mathsf{Alice}), cash(\gamma.\mathsf{Bob}))$): she adds x_a coins to γ.Alice's deposit in γ_a and x_b coins to γ.Bob's deposit in γ_b, as well as x_b coins in her own deposit in γ_a and x_a coins in her own deposit in γ_b.

In reality, unfortunately, there is no reason to trust γ.Ingrid that she will always be honest. In this case there are several problems with this procedure, probably the most obvious one being that γ.Alice and γ.Bob have no proof that they indeed sent their messages $\gamma_{\gamma.\mathsf{Alice}}$ and $\gamma_{\gamma.\mathsf{Bob}}$ in Step 1. It is also easy to see that such proofs need to have a special form. More precisely: they need to be such that later the parties can claim their money from γ.Ingrid, even if she misbehaves.

This is exactly where the contract *VPC* and the nanocontracts ν_a and ν_b will be used. Take for example γ.Alice (the case of γ.Bob is handled analogously). The contract *VPC* will be constructed in such a way that γ.Alice can send γ_a (defined in Step 1 above) to ν_a, and γ_a will be stored in the ν_a's state. Of course, as long as γ.Ingrid and γ.Alice are not in dispute, this can be resolved "peacefully": the parties would simply exchange their signatures on the new state (i.e., the new money distribution as described above). However, if the parties enter into a disagreement (this can only happen when one of them is corrupt), γ.Alice always has an option to execute ν_a "by force" via the blockchain. In parallel, γ.Bob will perform a symmetric procedure with Ingrid (for "sending γ_b"). The corresponding code for γ.Alice and γ.Bob is given in Step 2 and for γ.Ingrid in Step (3a) (Fig. 29). Note that such an execution requires pessimistic time T (where $T = 4 \cdot \Delta + 2$ is the time needed for the forced execution of a nanocontract), and optimistic time 2.

Party γ.Ingrid, after receiving (via the channel update, or forced execution) the message γ_a forwards it to γ.Bob by calling a corresponding function in ν_b(see Step 3a on Fig. 29) . This again is done by updating (or executing) the nanocontract ν_b. Symmetrically, γ.Ingrid forwards γ_b to γ.Alice. Moreover the contracts ν_p (for $p \in \{a, b\}$) will be constructed in such a way that once a contract receives both γ_a and γ_b it internally decides how to distribute money that it has in the same way as it is done in Step (2a) above (see case 1 in the code of $\mathsf{close}(\gamma^P)$ on Fig. 30).

Observe that "sending γ_p via a nanocontract ν_p" can take time at most T. After time T passed[19] and γ.Ingrid did not receive any message from γ.Alice (say) then γ.Ingrid can assume that γ.Alice will not send any message (this will happen only if γ.Alice is corrupt). In such a case γ.Ingrid calls a function finalize of ν_b (the case when γ.Bob sends no message is handled analogously). The purpose of this message is to "push" the contract to proceed (after the deadline T has passed). This is needed since the contracts never execute anything "by themselves", and always need an external message to start an execution

On the other hand γ.Ingrid should be guaranteed that she can forward γ_p to the other party even if he receives this message late (e.g. just before the "deadline" T). Hence we let the nanocontracts accept calls from γ.Ingrid also after time T, and in particular they can arrive to γ.Alice and γ.Bob in time at most $2 \cdot T$ (pessimistically). If γ.Alice or γ.Bob do not receive a message from γ.Ingrid within this time, then they call a function finalize() to the nanocontract to finish its execution.

Another technical thing that we need is to handle the situations when one of the parties does not send its message at all — for example, if γ.Ingrid does not forward any message to γ.Alice, or does not call finalize. Since this can happen only if γ.Ingrid is corrupt, we simply allow γ.Alice to call finalize() herself after time $2 \cdot T$ has passed (see Step 5 on Fig. 29), and take the money that was specified in γ_a away from γ.Ingrid (the same happens with Bob). This can take in the pessimistic case additionally T rounds. Hence, in total the pessimistic time is $3 \cdot T = 12 \cdot \Delta + 6$. This bound can be substantially reduced if we take into account the fact that the "state registration" procedure (which is a subroutine for the "forced execution") needs to be executed only once. We do not perform

[19] We count time from γ.validity.

a more careful analysis of the pessimistic running time, in order to keep our presentation as simple as possible. It is also easy to see that the standard running time is 2.

4.4 Discussion

There are several things that we omit in this paper, in order to simplify the exposition. In particular, we do not formally model the transaction fees (this approach is common in this area, see, e.g., [4, 2]). Let us emphasize that the transaction fees will be relevant only when the basic channels are open or closed (the virtual channels will have no fees, unless the intermediaries decide to provide their services at a fee). Although our system grantees that no party will be cheated by another one, it is still possible that some money will be lost on the transaction fees. Therefore the parties should open the channels only with the other users only if they can be relatively sure that these channels will not need to be closed immediately. In practice, we believe that in the standard case when Ingrid plays a role of a "bank" most of the transaction fees should be payed by the clients (especially in the situations when the clients have no confirmed identity or reputation).

We note that this problem exists in every channel network, and in our case it seems to be less severe, since the amounts of money claimed from the intermediaries will be larger (as they will "accumulate" several nanotransactions), and hence proportionally the fees will be lower than in the hash-locked-based solution.

Let us also remark that our construction implicitly assumes (as all the other payment channel constructions do) that there are no congestions on the blockchain. For example, a party can close several nanocontracts that it has, in a short period of time. In practice it should not be a big problem when a currency like Ethereum is used. Firstly, Ethereum has no official limit on the block size, and new blocks are appearing at a very high rate (so assuming a generous limit on transaction processing time should solve the problem). Secondly, Ethereum can process many messages that are sent to one contract in one block, so the nanocontracts can be closed in parallel (note that the order of executing different nanocontracts does not matter).

Another obvious problem is the need for permanent online availability by the parties, since they need to constantly monitor the network to see if the other party did not submit and old version of the channels. Again, this problem appears also in other payment networks, and solutions for this exists (e.g. network monitoring can be outsourced) [22].

Let us also say that we did not attempt to optimize the pessimistic execution times, since it would obscure the presentation of the main idea. However, some obvious optimizations can be done, for example, the first message that executes the nanocontract could be sent together with the first message of the "state registration" protocol.

One natural question is if one can have virtual channels that are (a) longer, (b) have a state. It turns out that this is possible, and in fact these two questions are closely related. More precisely: if one constructs virtual *state* channels, then one can apply our idea recursively for an arbitrary number of times. For example: suppose one wants to create a virtual state channel of length n among parties $P_1, \ldots, P_n$ (such that a basic state channel exists between each P_i and P_{i+1}). Then one can proceed as follows: (1) create a virtual state channel $P_1 \leftrightarrow P_3$ from basic channels $P_1 \Leftrightarrow P_2$ and $P_2 \Leftrightarrow P_3$, (2) create a virtual state channel $P_1 \leftrightarrow P_4$ from $P_1 \leftrightarrow P_3$ and $P_3 \Leftrightarrow P_4$, and so on (until the channel $P_1 \leftrightarrow P_n$) is created. This solution has pessimistic time complexity for channel creation and execution proportional to the length of the channel (where the unit is time Δ need to post transactions on a blockchain). However, it can be improved to constant time by letting the parties look into the blockchain and react appropriately to the conflicts resolved there publicly by other chain members (a similar idea has been recently presented for the Lightning-style channels in

Subprotocol for closing a virtual payment channel γ with identifier id when time $\gamma.\mathsf{validity}$ comes

Below for $P \in \gamma.\mathsf{end\text{-}users}$ let $\gamma^P := \Gamma^P(id)$ be P's local version of the virtual payment channel with identifier id. Moreover, define the following abbreviations:

- $nid_a := (\mathsf{Alice}, id)$,
- $nid_b := (\mathsf{Bob}, id)$,
- $id_a := \gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$, and
- $id_b := \gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$.

Procedure $\texttt{CloseStart}(id)$ executed by $P \in \gamma.\mathsf{end\text{-}users}$:

1. Erase γ^P from Γ^P.
2. Execute procedure $\texttt{OptimisticExecStart}$ with parameters:

$$(id_a, nid_a, \mathsf{close}, \Gamma^{\gamma.\mathsf{Alice}}(id)) \text{ if } P = \gamma.\mathsf{Alice}$$
$$(id_b, nid_b, \mathsf{close}, \Gamma^{\gamma.\mathsf{Bob}}(id)) \text{ if } P = \gamma.\mathsf{Bob}$$

   Go to Step 4.

Procedure $\texttt{CloseStart}(id)$ executed by $\gamma.\mathsf{Ingrid}$

3. Execute the following in parallel for

$$(ident, nid, ident', nid') := \begin{cases} (id_a, nid_a, id_b, nid_b) & \text{and} \\ (id_b, nid_b, id_a, nid_a) \end{cases}$$

   (a) For time at most $4\Delta+2$ wait for an output $(\mathsf{executed}, ident, nid, (\mathsf{closing}, \widehat{\gamma}))$ from $\texttt{OptimisticExecReact}$. Upon receiving such a message execute $\texttt{OptimisticExecStart}(ident', nid', \mathsf{close}, \widehat{\gamma})$.

   (b) If you do not receive the message in Step (3a) then execute

$$\texttt{OptimisticExecStart}(ident', nid', \mathsf{close}, \gamma_{\mathsf{init}}),$$

   where $\gamma_{\mathsf{init}} := \mathsf{virtual\text{-}channel}$, and

$$\texttt{OptimisticExecStart}(ident, nid, \mathsf{finalize}, ()).$$

Procedure $\texttt{CloseStart}(id)$ executed by $P \in \gamma.\mathsf{end\text{-}users}$:

4. For time $8\Delta + 4$ wait to receive from $\texttt{OptimisticExecStart}$ or $\texttt{OptimisticExecReact}$ an output

$$\begin{gathered}(\mathsf{executed}, id_a, nid_a, \mathsf{closed}, \gamma') \text{ if } P = \gamma.\mathsf{Alice} \\ (\mathsf{executed}, id_b, nid_b, \mathsf{closed}, \gamma') \text{ if } P = \gamma.\mathsf{Bob}\end{gathered} \tag{9}$$

5. If you do not receive the above output, then execute procedure $\texttt{OptimisticExecStart}$ with parameters:

$$(id_a, nid_a, \mathsf{finalize}, ()) \text{ if } P = \gamma.\mathsf{Alice}$$
$$(id_b, nid_b, \mathsf{finalize}, ()) \text{ if } P = \gamma.\mathsf{Bob}.$$

Fig. 29: The protocol description for closing a virtual channel γ when time $\gamma.\mathsf{validity}$ has come.

Function close(γ^P) **called by** $P \in$ end-users **after time** virtual-channel.validity **has passed**

Denote $\gamma_{\mathsf{init}} :=$ virtual-channel. If

$$\begin{array}{l}\text{current-time} > \gamma_{\mathsf{init}}.\text{validity} + T \quad \text{if } P \in \text{virtual-channel.end-users} \\ \text{current-time} > \gamma_{\mathsf{init}}.\text{validity} + 2T \text{ if } P = \text{virtual-channel.Ingrid}.\end{array}$$

then go idle. Otherwise check if γ^P is correctly signed by $P' := \gamma^P.\text{other-party}(P)$, if $\gamma^P.\text{total-cash} = \text{total-cash}$, and if $(\gamma.\text{id}, \gamma.\text{Alice}, \gamma.\text{Bob}) = (\gamma_{\mathsf{init}}.\text{id}, \gamma_{\mathsf{init}}.\text{Alice}, \gamma_{\mathsf{init}}.\text{Bob})$. If at least one of these conditions does not hold then go idle.
Otherwise consider two cases:

1. Your storage contains a value $(\widehat{P}, \widehat{\gamma}, \widehat{\tau})$ with $\widehat{P} = \text{other-party}(P)$ — in this case let:

$$\gamma^* := \begin{cases} \gamma^P \text{ if } \gamma^P.\text{ver-num} \geq \widehat{\gamma}.\text{ver-num} \\ \widehat{\gamma} \quad \text{otherwise.} \end{cases}$$

   For each $T \in$ end-users send a message closed together with $\gamma^*.\text{cash}(T)$ coins to T and terminate.
2. Otherwise store $(P, \gamma^P, \text{current-time})$ in your storage, send a message $(\text{closing}, \gamma^P)$ to both parties in end-users, and go idle.

Function finalize() **called by** $P \in$ end-users

If your storage contains a tuple $(\widehat{P}, \widehat{\gamma}, \widehat{\tau})$, such that

$$\begin{array}{l}\widehat{\tau} + \Delta < \text{current-time (in case } P \in \text{virtual-channel.end-users}), \text{ or} \\ \widehat{\tau} + 2\Delta < \text{current-time (in case } P = \text{virtual-channel.Ingrid})\end{array}$$

then for each $T \in$ end-users send a message closed together with $\widehat{\tau}.\text{cash}(T)$ coins to T and terminate. Otherwise ignore this call.

Fig. 30: Contract functions used in virtual channel closing.

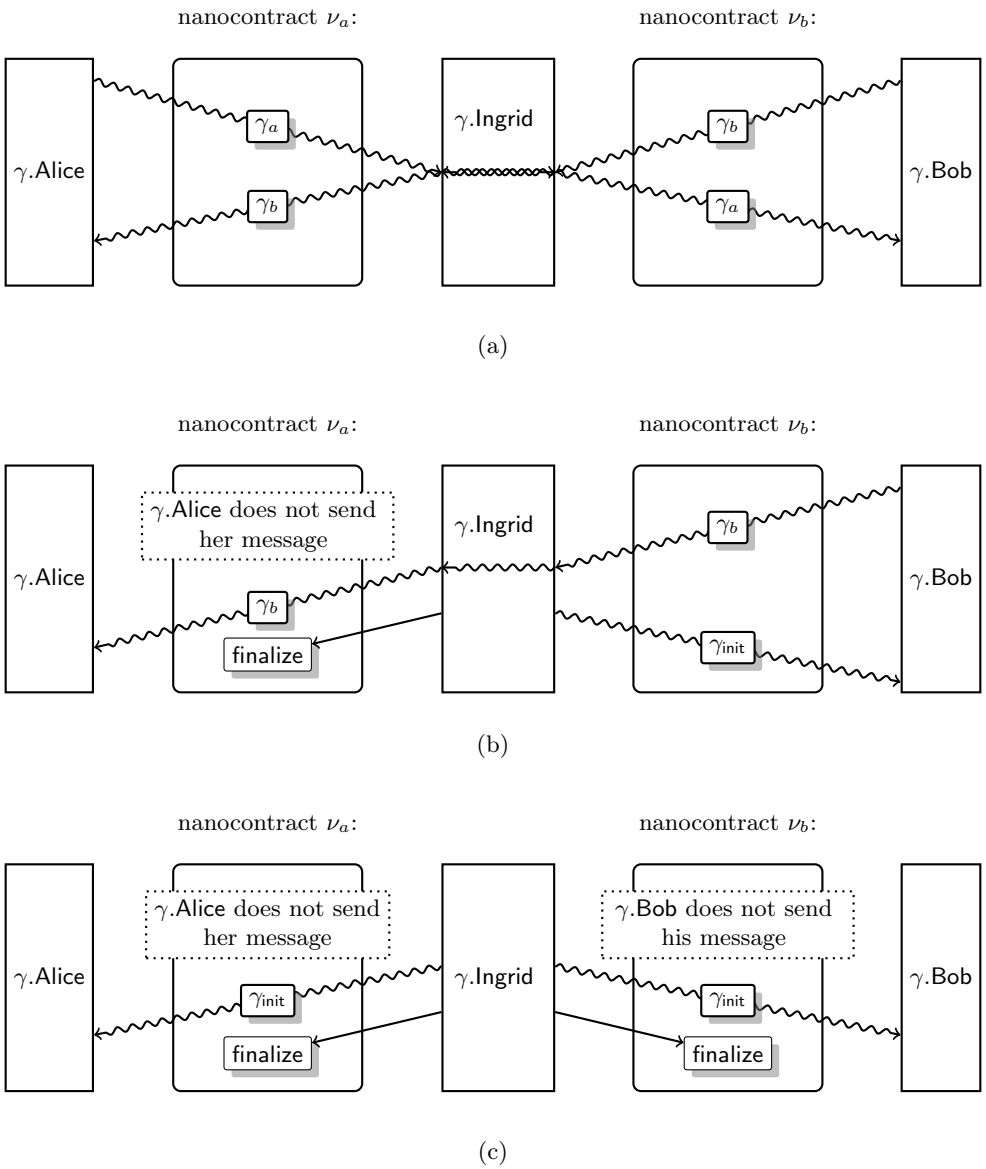


Fig. 31: The message flow in the virtual channel closing procedure. The snake arrows denote messages "passing through" the nanocontract. Channels γ_a and γ_b denote the channels that Alice and Bob (resp.) have with Ingrid, and γ_{init} is the initial version of the virtual channel.

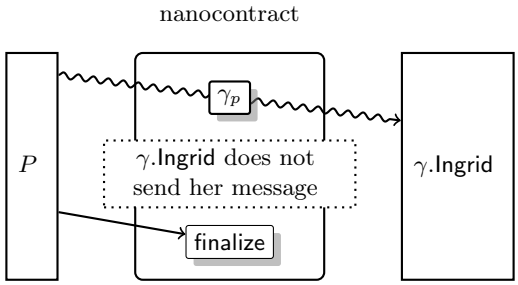


Fig. 32: The message flow in the virtual chanel closing procedure when Ingrid does not reply (γ_p is the version of γ stored by P).

[19]). Formally modeling this idea, constructing protocols, and proving security is beyond the scope of this paper. We are currently in the process of writing it down in a separate, subsequent paper.

Let us also mention, that our protocol can be improved to allow the parties to put money in the nanocontract execution (i.e. have a non-zero value as an argument for function Eval in Step 2, Fig. 17, page 31). This can be done by modifying the protocol in the following way: whenever a single nanocontract is registered on the ledger, all the other nanoconctracts in the same channel have to be also registered. As a result, the ledger becomes "aware" of the fact how much cash is left in the channel, and can decide if a user has enough coins in the channel to perform this execution.

Another obvious problem in our construciton is the fact that the intermediaries need to block the coins that are used for constructing the channels. This can be addressed by slightly relaxing the security guarantees. Namely, we can replace the full cheating-resilience (that has been assumed in this work), by a weaker "cheating-*evidence*". More precisely, the security guarantee in this case would be: "if an intermediary cheats then you can either get your money back, or you can post an evidence of this fact on the blockchain". Hence the risk that one gets cheated by an intermediary that has been functioning for a long time already is low, and probably acceptable in practice in case of nanotransactions.

References

[1] Ian Allison. *Ethereum's Vitalik Buterin explains how state channels address privacy and scalability*. https://tinyurl.com/n6pggct. 2016.

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. "Secure Multiparty Computations on Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. Berkeley, California, USA: IEEE Computer Society Press, 2014, pp. 443–458. DOI: 10.1109/SP.2014.35.

[3] Adam Back and Iddo Bentov. "Note on fair coin toss via Bitcoin". In: *CoRR* abs/1402.3698 (2014). URL: http://arxiv.org/abs/1402.3698.

[4] Iddo Bentov and Ranjit Kumaresan. "How to Use Bitcoin to Design Fair Protocols". In: *Advances in Cryptology – CRYPTO 2014, Part II*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2014, pp. 421–439. DOI: 10.1007/978-3-662-44381-1_24.

[5] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. "Instantaneous Decentralized Poker". In: *CoRR* abs/1701.06726 (2017). URL: http://arxiv.org/abs/1701.06726.

[6] *Bitcoin Wiki: Contract*. https://en.bitcoin.it/wiki/Contract. 2016.

[7] *Bitcoin Wiki: Payment Channels*. https://en.bitcoin.it/wiki/Payment_channels. 2016.

[8] *Bitcoin Wiki: Scalability*. https://en.bitcoin.it/wiki/Scalability. 2017.

[9] Jeff Coleman. *State Channels*. http://www.jeffcoleman.ca/state-channels/. 2015.

[10] Christian Decker and Roger Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Stabilization, Safety, and Security of Distributed Systems: 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*. Ed. by Andrzej Pelc and Alexander A. Schwarzmann. Cham: Springer International Publishing, 2015, pp. 3–18. ISBN: 978-3-319-21741-3. DOI: 10.1007/978-3-319-21741-3_1. URL: http://dx.doi.org/10.1007/978-3-319-21741-3_1.

[11] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: *Advances in Cryptology – EUROCRYPT 2015, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. Lecture Notes in Computer Science. Sofia, Bulgaria: Springer, Heidelberg, Germany, 2015, pp. 281–310. DOI: 10.1007/978-3-662-46803-6_10.

[12] Matthew Green and Ian Miers. *Bolt: Anonymous Payment Channels for Decentralized Currencies*. Cryptology ePrint Archive, Report 2016/701. http://eprint.iacr.org/2016/701. 2016.

[13] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. *TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub*. Cryptology ePrint Archive, Report 2016/575. http://eprint.iacr.org/2016/575, accepted to the Network and Distributed System Security Symposium (NDSS) 2017. 2016.

[14] Aniket Kate. "Introduction to Credit Networks: Security, Privacy, and Applications". In: *ACM CCS 16: 23rd Conference on Computer and Communications Security*. ACM Press, 2016, pp. 1859–1860.

[15] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. ISBN: 1584885513.

[16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2016, pp. 839–858. DOI: 10.1109/SP.2016.55.

[17] Joshua Lind, Ittay Eyal, Peter R. Pietzuch, and Emin Gün Sirer. "Teechan: Payment Channels Using Trusted Execution Environments". In: *CoRR* abs/1612.07766 (2016). URL: http://arxiv.org/abs/1612.07766.

[18] Silvio Micali and Ronald L. Rivest. "Micropayments Revisited". In: *Topics in Cryptology – CT-RSA 2002*. Ed. by Bart Preneel. Vol. 2271. Lecture Notes in Computer Science. San Jose, CA, USA: Springer, Heidelberg, Germany, 2002, pp. 149–163.

[19] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. "Sprites: Payment Channels that Go Faster than Lightning". In: *CoRR* abs/1702.05812 (2017). URL: http://arxiv.org/abs/1702.05812.

[20] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. http://bitcoin.org/bitcoin.pdf. 2009.

[21] Rafael Pass and Abhi Shelat. "Micropayments for Decentralized Currencies". In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel: Denver, CO, USA: ACM Press, 2015, pp. 207–218.

[22] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. Draft version 0.5.9.2, available at https://lightning.network/lightning-network-paper.pdf. 2016.

[23] Ronald L. Rivest. "Electronic Lottery Tickets as Micropayments". In: *FC'97: 1st International Conference on Financial Cryptography*. Ed. by Rafael Hirschfeld. Vol. 1318. Lecture Notes in Computer Science. Anguilla, British West Indies: Springer, Heidelberg, Germany, 1997, pp. 307–314.

[24] Ethereum Team. *Solidity Documentation, Release 0.4.11*. avaiable at https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf. 2017.

[25] *Update from the Raiden team on development progress, announcement of raidEX*. https://tinyurl.com/z2snp9e. 2017.

[26] David Wheeler. "Transactions Using Bets". In: *Proceedings of the International Workshop on Security Protocols*. London, UK, UK: Springer-Verlag, 1997, pp. 89–92. ISBN: 3-540-62494-5. URL: http://dl.acm.org/citation.cfm?id=647214.720381.

[27] *Wikipedia: Microtransaction*. https://en.wikipedia.org/wiki/Microtransaction. 2017.

[28] Gavin Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. http://gavwood.com/paper.pdf. 2016.

A Solidity Contracts

We implemented our contracts over Ethereum. The code of all the contracts and a simple user interface is available at `https://github.com/VirtualPaymentChannels2017/VPM2017` (currently the code handles just one nanocontract in the state channel). To implement the state channels in Ethereum we need two contracts created on the blockchain (plus a singleton library contract LibSignatures that provides the signature verification functionality). The main contract is a *MSContract* which internally requires another contract, the Virtual Payment Contract (*VPC*). Below we provide some explanations of this code.

MSContract The state channel is initiated by one of the parties by creating the *MSContract* with the addresses of Alice and Bob. Next, the parties call the confirm function which is used to send funds to the contract during a predefined timeout period after channel creation. Should one of the parties not call this function in time, the other party's money is refunded (by calling the refund function). Otherwise, the *MSContract* is created and can be closed if both parties call the closeChannel function (or only one of them if the other party disappears).

Virtual Payment Contract If both parties have a *MSContract* connecting them to a third party, they can add a virtual payment contract *VPC* to their *MSContract* contracts. Ideally, this is done without interaction with the blockchain. But in order to create such a link they need an address of a *VPC* contract which means that this contract needs to be created on the blockchain at this point. Note that this contract needs to be created only once and then it can be used for all possible virtual channels (between any parties) so most likely it has already been created. Should some problem occur, the parties first establish the states of the *MSContract* contracts using the function stateRegister which fixes the blocked money, address and parameters for the *VPC*. Then, the parties spanning up the virtual channel need to run the exec functionality of the *VPC* contract. After the *VPC* machine is executed and its state is settled, one of the parties calls the function execute in the *MSContract* which internally queries the state of *VPC*. Only if this function returns a distribution of the blocked money, the State Channels can be closed. The settlement of state, execution of the machine and closing of the channel can always be executed by one party alone (Alice, Bob or Ingrid).

A.1 Transaction costs

The transaction costs are presented on Fig. 33. As expected, the most expensive operations are the creation of *MSContract* and *VPC*. However, the *VPC* has to be created only once and can be reused in all the virtual channels, so its cost is not a problem. On the other hand *MSContract* is destroyed when the parties close the channel between them and some of the gas used in the creation procedure is returned to Alice.

We note that we did not attempt to optimize transaction costs, and therefore they are likely to be lower if some optimization is performed.

operation	gas	ETH
LibSignatures create	325149	0.00650298
MSContract create	1981539	0.03963078
MSContract confirm	44287	0.00088574
MSContract refund	11456	0.00022912
MSContract stateRegister	213241	0.00426482
MSContract finalizeRegister	34214	0.00068428
MSContract execute	43372	0.00086744
MSContract close	78688	0.00157376
MSContract finalizeClose	23005	0.00046010
VPC create	1217912	0.02435824
VPC exec	399299	0.00798598
VPC finalize	40061	0.00080122

Fig. 33: Transaction costs for concrete operations.