PERUN: Virtual Payment Hubs over Cryptographic Currencies

Stefan Dziembowski
University of Warsaw

Lisa Eckey
TU Darmstadt

Sebastian Faust
TU Darmstadt

Daniel Malinowski
University of Warsaw

***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*, an off-chain channel system that offers a novel 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, for each individual payment, involvement of the intermediary over which the payment is routed. This is achieved by introducing a new technique called "virtual payment channels". In this paper we formally model and prove security of this technique for routing payments over one intermediary, who can be viewed as a "payment hub" that has direct channels with several parties. Our scheme works over any cryptocurrency that provides Turing-complete smart contracts. As a proof of concept, we implemented Perun's smart contracts in *Ethereum*.**

I. Introduction

Decentralized cryptocurrencies, such as Bitcoin or Ethereum, have gained great popularity over the last 10 years. They exist within an open, decentralized peer-to-peer network and provide a payment infrastructure without any central authority regulating transactions. In addition, cryptocurrencies have helped to accelerate deployment of disruptive technology such as smart contracts, which use program code to enforce complex agreements. The core technological innovation empowering decentralized cryptocurrencies is a consensus mechanism for maintaining a distributed ledger – the so-called *blockchain*. The blockchain is an append-only register for irreversibly storing the system's transactions. Because the entire state of the blockchain is replicated among thousands of users, the number of transactions and the speed at which they are processed is limited when compared to centralized systems. For instance, the most prominent blockchain-based cryptocurrency Bitcoin comes with a built-in limitation of processing up to 7 transactions per second and requires on average 10 minutes to confirm new transactions.

The scalability problems of blockchain-based cryptocurrencies are drastically amplified with the emergence of microtransactions that require users to transfer small amounts of money between each other and can, e.g., be used for sharing WiFi or pay per drive insurance models. Typically such microtransactions have to be executed instantaneously, which is a problem in ledger-based cryptocurrencies, where confirmation can take up to several minutes. Moreover, posting transactions on the ledger results into fees, which usually are much higher than the value of a microtransaction. Therefore, it seems unlikely that current cryptocurrencies can directly support microtransactions, and the many applications they offer.

An exciting proposal to address the above challenges is a technology called *payment channels* [3], which allows two parties to rapidly exchange money between each other via so-called off-chain" transactions. In contrast to on-chain transactions, off-chain transactions enable users to exchange money without directly interacting with the ledger (except when the channel is opened or closed). The concept of payment channels has been extended to so-called payment *networks*, which enable users to route transactions via intermediary hubs. This has the advantage that channels can be re-used, thereby further decreasing the on-chain transaction load. An example of such a network has been designed in [16] over Bitcoin. In this system the payments are routed over the network in the following way. Suppose two parties, Alice and an *intermediary* called Ingrid, established a channel (denote it: "β_A"), and Ingrid also has a channel with Bob, denoted "β_B" (but Alice and Bob do not have a payment channel between each other). In other words Ingrid can be viewed as a payment hub at which Alice and Bob have "accounts". Then Alice can perform a micropayment, for y coins to Bob via Ingrid. In [16] each such money transfer requires explicit confirmation by Ingrid. This introduces latency and additional costs.

A. Summary of our contribution and its applications.

The main contribution of this work is addressing the aforementioned shortcoming with a concept that we call *virtual channels* (abbreviated: vc). Again, suppose Alice and Bob are both connected by a channel created over the blockchain with an intermediate payment hub Ingrid, but they do not have a direct channel between each other. We will call such channels that are built directly over the ledger in the following *ledger channels* (lc). A virtual channel establishes a direct (virtual) link between Alice and Bob, where the intermediary Ingrid does not need to get involved in each payment. This significantly reduces latency and costs, and moreover is beneficial for privacy, because Ingrid cannot observe the individual money transfers between Alice and Bob[1]. We call our system

[1]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, [9, 10]).

Perun[2]. We provide a full formal specification of the Perun virtual channel system, formalize its security properties using ideal/real world paradigm in a style of the UC framework of Canetti [4] and prove that our protocol satisfies our security definition.

While our system works over any cryptocurrency which allows Turing complete smart contracts (we give a short introduction to this concept in Sect. III-A1), we demonstrate the feasibility of our proposal by providing a prototype implementation of the contracts underlying the Perun channel system in *Solidity* (see Appx. A), one of the main languages supported by the Ethereum cryptocurrency [7].

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 ledger 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. 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). Further related work is described in Appx. B.

B. Notation.

We 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 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. $T(attr)$) is referred to as $T.attr$. This convention allows us to easily handle tuples that have dynamically 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 an additional attribute $attr$ and $T'.attr = x$. We sometimes define a function $f : \{x_1, \ldots, x_n\} \to \mathcal{Y}$ by providing its function table in a form $f = [x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ (meaning that for every i it holds that $f(x_i) = y_i$). In our security definition we use the notion of *computational indistinguishability* of distribution ensembles (see, e.g., [8]). Whenever we say that some operation (e.g. delivering a message, changing state of an ideal functionality, or simply staying idle) *takes time at most* $\tau \in \mathsf{time}$ we mean that it is up to the adversary (or the simulator, in the ideal world) to decide how long this operation takes (as long as it takes at most τ rounds). We denote an empty string with ε.

[2]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).

II. Perun's informal description

In this section we informally describe Perun's functionality, i.e., what properties and functionality it provides. Its formal specification appears in Sect. III, and the protocol that implements it is given in Sect. IV.

A. Ledger channels

We start with a description of the standard "ledger" payment channels [16], which are created by interacting with the blockchain, and allow two parties to instantaneously carry out payments between each other. More precisely, the ledger is used only when parties involved in the payment channel disagree, or when they want to open/close the channel. As long as the parties are not in conflict, they can freely *update* the balance of the channel (i.e. transfer coins between each other's accounts). At a high-level a ledger payment channel β between two parties, Alice and Bob, starts with an *opening procedure*, where Alice deposits x_A coins into the channel and Bob deposits x_B coins respectively (for some $x_A, x_B \in \mathbb{R}_{\geq 0}$). Until channel β is closed, these coins remain "blocked", i.e., the parties cannot use them for any other purpose. Initially, the *balance* of the channel can be described by a function defined as

$$[\text{Alice} \mapsto x_A, \text{Bob} \mapsto x_B] \tag{1}$$

meaning that Alice "has x_A coins in it" and Bob "has x_B coins in it", and the *value* of the channel is $x_A + x_B$. After this set-up has been completed, Alice and Bob can *update* the distribution of the funds in the channel multiple times without interacting with the blockchain. The update mechanism is used for performing payments between Alice and Bob. If, for example, Alice wants to pay some amount $w \leq x_A$ of coins to Bob, then the parties perform an update that changes the balance of β to

$$[\text{Alice} \mapsto (x_A - w), \text{Bob} \mapsto (x_B + w)].$$

At some point one of the parties that opened the channel can decide to *close* it. If, for example, Alice wants to close the channel, she commits the current balance $[\text{Alice} \mapsto x'_A, \text{Bob} \mapsto x'_B]$ of the channel to the blockchain and the funds are distributed accordingly to Alice and Bob (i.e. Alice and Bob receive x'_A and x'_B coins respectively).

B. Virtual channels

The main novelty of Perun is a new method for connecting ledger channels that is alternative to "payment routing" used in existing payment channel networks. Namely, Perun offers so-called "virtual channels" that minimize the need for interaction with the intermediaries in the channel chains, and in particular do not require that intermediates confirm individual payments routed via them. The basic idea of virtual channels is to apply the channel technique recursively, by building a virtual payment channel "on top of" the ledger channels.

To better illustrate the concept of virtual channels, consider three parties, Alice, Bob and Ingrid, and suppose that there exists a ledger channel β_A between Alice and Ingrid, and β_B between Ingrid and Bob with the following balances:

$$\begin{array}{ll} \beta_A: & [\text{Alice} \mapsto y_A, \text{Ingrid} \mapsto y_I] \\ \beta_B: & [\text{Ingrid} \mapsto z_I, \text{Bob} \mapsto z_B]. \end{array}$$

In Perun, Alice and Bob can establish a *virtual* payment channel γ with the help of Ingrid (using channels β_A and β_B), and *without* touching the ledger. Suppose that the initial balance of γ after opening is denoted by as on Eq. (1). After γ is open, the balances of β_A and β_B become:

$$\begin{array}{ll} \beta_A: & [\text{Alice} \mapsto (y_A - x_A), \text{Ingrid} \mapsto (y_I - x_B)] \\ \beta_B: & [\text{Ingrid} \mapsto (z_I - x_A), \text{Bob} \mapsto (z_B - x_B)]. \end{array} \quad (2)$$

Opening of γ is possible only if all the values above are non-negative, i.e., $x_A \leq \min(y_A, z_I)$ and $x_B \leq \min(y_I, z_B)$. In other words: Alice, Bob, and Ingrid need to have enough coins in the ledger channels to open γ. These coins remain blocked in β_A and β_B as long as the virtual channel is open. For Alice and Bob this is similar to the situation when the coins are blocked on the ledger in a newly created ledger channel. What can be viewed as a disadvantage is that also Ingrid has to block her coins. In Appx. C we discuss a solution for this problem that is based on slightly weakening the security guarantees. Once a virtual channel is opened, it can be updated multiple times, exactly in the same way as the ledger channel, i.e., transferring w coins from Alice to Bob results in a new balance of γ as before. As long as everybody is honest, Alice and Bob need to interact with Ingrid only when the channel is opened and when it is closed, and in particular each update of γ does *not* require interacting with Ingrid (in the same way as updating a ledger channel does not require interacting with the ledger).

The "financial consequences" of closing a virtual channel appear on the ledger channels β_A and β_B (and not directly on the blockchain, as it is the case for the ledger channels). For example, suppose that the current balances of the ledger channels β_A and β_B are $[\text{Alice} \mapsto y'_A, \text{Ingrid} \mapsto y'_I]$ and $[\text{Ingrid} \mapsto z'_I, \text{Bob} \mapsto z'_B]$, respectively (note that this may be different from the balance on Eq. (2) as the ledger channels β_A and β_B could have been updated in the meanwhile), and the current balance of γ is $[\text{Alice} \mapsto x'_A, \text{Bob} \mapsto x'_B]$. Then closing γ results in the following balances of the ledger channels:

$$\begin{array}{ll} \beta_A: & [\text{Alice} \mapsto (y'_A + x'_A), \text{Ingrid} \mapsto (y'_I + x'_B)] \\ \beta_B: & [\text{Ingrid} \mapsto (z'_I + x'_A), \text{Bob} \mapsto (z'_B + x'_B)]. \end{array} \quad (3)$$

Observe that for $P \in \{\text{Alice}, \text{Bob}\}$ the financial consequences of all the operations on γ are exactly as one might expect, i.e. P's net financial result is that she gains $x'_P - x_P$ coins in her balance in β_P (where "gaining q" coins means loosing $-q$, if q is negative). On the other hand observe that the consequences for Ingrid are "neutral", i.e., if she gains z coins in β_A then she looses the same amount in β_B (and vice versa). We will require that, as long as γ is open, the ledger channels β_A and β_B cannot be closed. In other words, the parties that opened these ledger channels have to wait with closing them until the financial consequences from closing of channel γ are known. One problem with this is that that Ingrid should be sure that her coins do not get blocked in β_P for a very long period of time (or: forever). This is different from the ledger channels, where the role of the "intermediary" is played by the blockchain, which does not have "her own coins" invested in the protocol.In particular it is completely ok "from the point of view of the ledger" if a ledger channel is never closed. For this reason the virtual channels come with a special attribute called *validity* that Alice, Bob and Ingrid agreed upon when the virtual channel was opened. A virtual channel is closed when its validity expires (note that this is different from the ledger channels, where closing is initiated by Alice or Bob). Thanks to this solution Ingrid can be sure that she gets her coins back after some period of time. Another (slightly more complicated) option would be to allow Ingrid to request virtual channel closing at any time.

Let us emphasize that our scheme is secure against arbitrary corruptions of Alice, Ingrid, and Bob. and in particular, no assumption about the honesty of Ingrid is needed. There are several subtle problems that need to be solved when designing protocols that deal with such strong corruption models. Some of them come from concurrency. Our solution achieves security in a fully concurrent setting, i.e., several ledger and virtual channels can be opened and closed simultaneously. Security of our solution relies on smart contracts that are used to build channels β_A and β_B. One main challenge is to design protocols that minimize the use of these smart contracts, and hence, we require access to them only during final settlement. Moreover, we have taken special care to reduce the amount of data that needs to be processed by these contracts (e.g. it does *not* grow with time of the protocol execution). For simplicity of exposition we do not model the transaction fees (we address them, however, when we talk about the implementation, see Appx. A).

III. Formal model and security definition

To analyze the security of our solution, we use the simulation-based security definition inspired by the UC framework [4], in which security is defined by comparing two worlds: the *real* and the *ideal* world. At a high level the ideal

world gives a formal description of the desired specification of a protocol using a concept called an *ideal functionality* (in our case this functionality is called Channels and is described in detail in Sect. III-C). In the real world the parties run a protocol that implements this functionality. The protocol that we construct will be denoted by `channels` and will run among parties from the set $\mathcal{P} = \{P_1, \ldots, P_n\}$, which are modeled as interactive machines. The P_i's in the real world are connected by secure (secret and authentic) communication channels. A protocol is executed in a presence of an *adversary* $\mathcal{A}$ (in the ideal world called a *simulator* $\mathcal{S}$) and an *environment* $\mathcal{Z}$, which are both also interactive machines. The adversary can *corrupt* any party P_i, by which we mean that he takes full control over P_i. The environment is responsible for providing inputs to the parties, and acts as a distinguisher between the real and the ideal world. The latter means that if there exist no PPT environment that can distinguish between the real and the ideal execution, then the protocol is deemed secure.

In addition to the above entities, the parties running the protocol have access to some ideal functionalities: the *contract* functionality $\mathcal{C}$ (in the real world), the *channels* functionality Channels (in the ideal world) and the *financial ledger functionality* $\mathcal{L}$ (in both worlds). We describe the functionalities $\mathcal{C}$ and Channels in Sections III-A1 and III-B, respectively. Here, we will describe briefly the financial ledger functionality $\mathcal{L}$ because it is responsible for handling coins. The financial ledger functionality interacts only with the environment, and the functionalities $\mathcal{C}$ and Channels (note that it does not interact directly with the parties, but $\mathcal{C}$ and Channels are interfaces to it). We assume that the financial ledger functionality is *global* [5], i.e., there is only a single such ideal functionality and that its state is "visible" for the environment (i.e. the environment is informed about all the messages that $\mathcal{L}$ receives and sends). It is initiated by the environment $\mathcal{Z}$ that preloads the parties with coins on $\mathcal{L}$. The financial operations are performed via the ideal functionalities Channels and $\mathcal{C}$ who can *add y coins to* (or *remove y coins from*) P's account on the ledger (these commands are executed instantaneously). In the ideal world we allow the simulator $\mathcal{S}$ to freely remove money from the accounts of corrupt parties and to add them (with an arbitrary delay) to the accounts of other (corrupt or honest) parties. This corresponds to the fact that we are not interested in preventing the corrupt parties from "acting irrationally" and loosing money. The financial ledger functionality is formally depicted on Fig. 4 in Appx D.

When we say that a party P sends a message m to an ideal functionality $\mathcal{I}$ (where $\mathcal{I}$ is either $\mathcal{C}$ or Channels) *together with x coins* it means that when m arrives to $\mathcal{I}$, the functionality $\mathcal{I}$ removes x coins from P's account in $\mathcal{L}$ (if P does not have sufficient amount of coins then message m is ignored). Similarly $\mathcal{I}$ sends a message m' to P *together with x' coins* means that x' coins are added to P's account in $\mathcal{L}$.

We assume a synchronous communication network, i.e., the execution of the protocol happens in rounds. The parties, the ideal functionalities, the environment and the adversary are always aware of a given round. Let $\mathsf{time} := \mathbb{N}$ denote the set of all possible round numbers. We assume that if in round i a party sends a message to another party (or an ideal functionality), then it arrives to it at the beginning of round $i + 1$. The adversary can decide about the order in which the messages arrive in a given round, but we assume that he cannot change the order of messages sent between two honest parties (this can be easily achieved by using, e.g., message counters). For simplicity we assume that computation takes no time and is "atomic". The communication between adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ takes no time. Sending messages to the ledger and the contract functionalities takes between 1 and Δ rounds (we assume that this includes also time needed by the parties to receive the message back from the contract). Messages send by these functionalities arrive to all the parties in 1 round.

Since we consider stand-alone security we do not use the session identifiers ("*sid*" in the UC terminology). We also do not use notation like "*ssid*" — instead we simply say that a party (or a contract) "replies to a message". We believe that omitting these technical notation improves readability of our protocols. We also use a concept of *threads* that should be understood as particular instances of the protocol. For example each ledger channel will have a corresponding "thread" in the protocol for its users. Messages sent between all the entities will start with a `keyword` in typewriter font. The keywords in messages exchanged with the environment will additionally be underlined.

A. The real world execution

To simplify exposition we assume that before the protocol starts a public-key infrastructure setup phase is executed by some trusted party. The signature of P on m will be denoted $\mathsf{Sign}_P(m)$. We emphasize that the use of a PKI is only an abstraction that helps to describe our protocols. In practice, the trusted setup can, e.g., easily be realized using the blockchain. Parties will often use their secret key to sign and verify messages. In this case, we say that a tuple $(x_1, \ldots, x_n, \sigma)$ is *signed by* P if σ is a valid signature of P on $(x_1, \ldots, x_n)$.

1) Modeling smart contracts. We make use of smart contracts (see, e.g, [2, 11]), 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. Such contracts are used to resolve disputes between users about the channel's state, or simply to close the channel. On an intuitive level one can think of a contract as an independent entity that receives coins and messages from the parties and sends coins and messages back to them. We model the contracts using a *contract functionality* $\mathcal{C}$ that is present only in the real world (In UC terms one would say that our protocol is designed in the $\mathcal{C}$ hybrid world). It interacts with the parties in $\mathcal{P}$ and with the financial ledger functionality $\mathcal{L}$, and is part of the protocol that we construct (see Fig. 3). It maintains a set of contract *instances* (i.e. the individual contracts created by the parties). More details on our concrete contract functionality $\mathcal{C}$ appear in Sect. IV.

B. *The ideal functionality* Channels

Before describing the Channels functionality let us first define more formally the channel syntax. To this end, we first introduce two types of functions for specifying the current balance of a channel and for handling transfers between parties. We say that $balance$ is a *balance function for parties* $P, P' \in \mathcal{P}$ if its type is $\{P, P'\} \to \mathbb{R}_{\geq 0}$. We also say that θ is a *transfer function for parties* P *and* P' if its type is $\{P, P'\} \to \mathbb{R}$ and $\theta(P) + \theta(P') = 0$ (note that usually one of the values $\theta(P)$ and $\theta(P')$ is negative). These functions can be added in a natural way, i.e., if f and g are transfer or balance functions for P and P', then $h = f + g$ is a function $h : \{P, P'\} \to \mathbb{R}$ defined as $h(P) := f(P) + g(P)$ and $h(P') := f(P') + g(P')$.

If $f : \{P, P'\} \to \mathbb{R}$ is a balance or transfer function then $f(P)$ is called the *amount of coins that* P *has in* f. If function $g : \{P, P'\} \to \mathbb{R}_{\geq 0}$ is a balance function, then *adding* $x \in \mathbb{R}$ *coins to the account of* P *in* g results in a function $g' : \{P, P'\} \to \mathbb{R}_{\geq 0}$ equal to $g(P')$ on input P', and to $g(P) + x$ on input P. *Removing* x *coins* is a shorthand for "adding $-x$ coins". If $h : \{P, P'\} \to \mathbb{R}_{\geq 0}$ is a transfer function, then *transferring* $x \in \mathbb{R}$ *coins from the account of* P *to the account of* P' *in* h results in a function $h' : \{P, P'\} \to \mathbb{R}_{\geq 0}$ equal to $h(P') + x$ on input P', and to $h(P) - x$ on input P.

We define a *ledger channel over a set of parties* $\mathcal{P}$ as an attribute tuple β of the form:

$$\beta = (\beta.\mathsf{id}, \beta.\mathsf{Alice}, \beta.\mathsf{Bob}, \beta.\mathsf{balance})$$

and a *virtual payment channel* γ *over a set of players* $\mathcal{P}$ as an attribute tuple of the form:

$$\gamma = (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Ingrid}, \gamma.\mathsf{Bob}, \gamma.\mathsf{balance}, \gamma.\mathsf{subchan}, \gamma.\mathsf{validity}).$$

For a ledger/virtual channel δ the value $\delta.\mathsf{id} \in \{0,1\}^*$ is called the *identifier of* δ and $\delta.\mathsf{Alice}, \delta.\mathsf{Bob}$ are two distinct elements of $\mathcal{P}$. For a virtual channel $\gamma.\mathsf{Ingrid}$ is also an element of $\mathcal{P}$ (distinct from $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$) and it is sometimes called the *intermediary*. We define the set *end-users of* δ as $\delta.\mathsf{end\text{-}users} = \{\delta.\mathsf{Alice}, \delta.\mathsf{Bob}\}$ (note that when δ is a virtual payment channel, then this set does not contain $\delta.\mathsf{Ingrid}$). We also say that δ *is established between the parties in* $\delta.\mathsf{end\text{-}users}$. We also define the shortcut $\delta.\mathsf{other\text{-}party} : \delta.\mathsf{end\text{-}users} \to \delta.\mathsf{end\text{-}users}$ as $\delta.\mathsf{other\text{-}party}(\delta.\mathsf{Alice}) = \delta.\mathsf{Bob}$ and $\delta.\mathsf{other\text{-}party}(\delta.\mathsf{Bob}) = \delta.\mathsf{Alice}$. If δ is a virtual channel then $\delta.\mathsf{all\text{-}users}$ denotes the set $\{\delta.\mathsf{Alice}, \delta.\mathsf{Bob}, \delta.\mathsf{Ingrid}\}$, and if δ is a ledger channel then simply $\delta.\mathsf{all\text{-}users} = \delta.\mathsf{end\text{-}users}$. The attribute $\delta.\mathsf{balance}$ is a balance function for parties $\delta.\mathsf{end\text{-}users}$.

Let us now take a look at the additional attributes of a virtual channel. One of the additional attributes that virtual channels have over ledger channels is the function subchan. For a virtual channel γ we have that $\gamma.\mathsf{subchan}$ is a function from $\gamma.\mathsf{end\text{-}users}$ to $\{0,1\}^*$ with the following definition. For $P \in \gamma.\mathsf{end\text{-}users}$ the value $\gamma.\mathsf{subchan}(P)$ will be called the *identifier of* P*'s subchannel of the virtual channel* γ, and it will be used to indicate the corresponding identifier of the ledger channel that is used to *construct* γ. Finally $\gamma.\mathsf{validity} \in \mathsf{time}$ denotes the *channel validity*, i.e., the round until which the virtual payment channel stays open.

In the ideal world the parties do not execute any protocol. Instead they simply receive messages from the environment and forward them to the ideal functionality Channels. They forward the replies that they receive from Channels to the environment. The corrupt parties may of course deviate from the specification. The ideal functionality Channels is presented in full detail on Fig. 5 in Appendix (page 17). Below we provide a slightly informal description of its functionality due to space limitations.

Functionality Channels maintains a *channel space*, which is a set Σ that consists of some ledger and virtual channel tuples. We will assume that for every id there exists at most one channel $\delta \in \Sigma$ such that $\delta.\mathsf{id} = id$ (we will also refer to such channel as $\Sigma(id)$). We require that for every virtual channel $\gamma \in \Sigma$ there exist ledger channels $\beta_a, \beta_b \in \Sigma$ such that $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice}) = \beta_a.\mathsf{id}$ and $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob}) = \beta_b.\mathsf{id}$, i.e., the channels β_a and β_b that were used to construct γ also belong to Σ. Initially Σ is empty. The functionality Channels consists of the following parts.

1) Opening a ledger channel. A ledger channel β between two parties $\beta.\mathsf{Alice}$ and $\beta.\mathsf{Bob}$ is created in an opening procedure, in which $\beta.\mathsf{Alice}$ puts $x_A := \beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins into the channel and $\beta.\mathsf{Bob}$ puts $x_B := \beta.\mathsf{balance}(\beta.\mathsf{Bob})$ coins into it. Hence, initially $\beta.\mathsf{Alice}$ and $\beta.\mathsf{Bob}$ have x_A and x_B coins (respectively) in the channel (and the total value of β will always be $x_A + x_B$). We assume that the opening of channel β is always initiated by party $\beta.\mathsf{Alice}$, who sends to Channels a message $(\underline{\texttt{lc-open}}, \beta)$ together with x_A coins. These coins are removed from $\beta.\mathsf{Alice}$'s account on the ledger. The functionality waits time Δ to receive the same message from $\beta.\mathsf{Bob}$ together with x_B coins. If this message is received then the channel is opened, and β is added to Σ (which is communicated to $\beta.\mathsf{end\text{-}users}$ by a message $(\texttt{lc-opened})$). Otherwise in time at most 2Δ the functionality sends to $\beta.\mathsf{Alice}$ a message $(\texttt{lc-not-opened})$ and refunds her the coins that were removed from her account.

2) Opening and closing a virtual channel. As explained in Sect. II-B the virtual channels come with a special attribute "validity" that specifies when a given channel will be closed. Since they cannot be closed earlier, there is no separate "closing" procedure for the virtual channels, and instead both opening and closing procedure are described together.

Opening happens when the parties from $\gamma.\mathsf{all\text{-}users}$ send a message $m = (\underline{\texttt{vc-open}}, \gamma)$ to the functionality Channels. When they are all honest this will happen in the same round τ (due to the restrictions on the environment that we describe later, see Sect. III-C, Point 5), but if some of them are dishonest, we allow time difference of at most 2 rounds. If some of the parties in $\gamma.\mathsf{all\text{-}users}$ did not send m to the functionality, then the channel is not created, and the functionality informs the parties about it by sending a message $(\underline{\texttt{vc-not-opened}})$ to them. Otherwise, if the functionality receives m from

all the parties in γ.all-users, then let $\beta_P := \gamma.\mathsf{subchan}(P)$ for $P \in \gamma$.end-users. The functionality removes the coins from the ledger channels according to the rules presented informally in Sect. II-B (see Eq. (2)), i.e, for both P it removes $\gamma.\mathsf{balance}(P)$ coins from P's account in β_P, and $\gamma.\mathsf{other\text{-}party}(P)$ coins from γ.Ingrid account in β_P. Then it adds γ to Σ, and informs the parties in γ.all-users that the channel has been opened by sending a message ($\underline{\texttt{vc-opened}}$) to them.

After the channel is opened the functionality accepts the channel update requests from γ.end-users (this is described in Sect. III-B3 below). As described in Sect. II-A when time $\gamma.\mathsf{validity}$ comes the channel γ is closed "automatically". Let $\widehat{\gamma}$ be the current state of channel γ. Within time $\gamma.\mathsf{validity} + 5\Delta + 1$ the coins that the γ.end-users have in γ are credited to γ.all-users accounts in the subchannels from $\gamma.\mathsf{subchan}$ according to the rules described in Sect. II-B (see Eq. (3)), i.e., for each $P \in \gamma.\mathsf{end\text{-}users}$ party P gets $\widehat{\gamma}(P)$ coins to her account in β_P, and γ.Ingrid gets $\widehat{\gamma}.\mathsf{other\text{-}party}(P)$ coins to her account in β_P. Then the functionality sends a message ($\underline{\texttt{vc-closed}}$) to $\gamma.\mathsf{all\text{-}users}$, and $\widehat{\gamma}$ is erased from Σ.

3) Ledger/virtual channel update. Once a channel is open, the parties can *update* the distribution of funds in the channel without interacting with the blockchain. A channel can be updated multiple times, subject only to the restriction that the value of the channel remains unchanged. Since this procedure is identical for ledger and virtual channels, we denoted the channel that is updated with a letter δ (the reader should keep in mind that δ can be either a ledger channel β or a virtual channel γ). Let θ be a transfer function (see Sect. III-B) for $\delta.\mathsf{end\text{-}users}$, and let $\alpha \in \{0,1\}^*$ be a string called an *update annotation* that provides a way for the parties to agree on why a given transfer happens (its role is similar to the "purpose of payment description" in the wire bank transfer). An *update that performs a transfer θ on channel δ*, is *annotated with a string α* and is *initiated by P* (called an an *initiator*). It starts when P receives a message ($\underline{\texttt{update}}, id, \theta, \alpha$) (where id is the identifier of δ in Σ). As a result of this, P' (the *confirmer*) asks the environment if this update is ok by sending a "($\underline{\texttt{update-requested}}, id, \theta, \alpha$)" message to it. Once the environment agrees (by sending a "$\underline{\texttt{update-ok}}$" message to P') the channel δ is updated, i.e., it is replaced in Σ by $\widehat{\delta}$ that is equal to δ, except that $\widehat{\delta}.\mathsf{balance} := \delta.\mathsf{balance} + \theta$. If the above steps finish in 2 rounds, then we say that an update was *successful*, otherwise we say it *failed*.

4) Ledger channel closing. As described in Sect. II-A a ledger channel β can be closed by any party $P \in \beta.\mathsf{all\text{-}users}$. This can be done at any time, provided there are no virtual channels open over β. Closing of a channel β is initiated by sending a message ($\underline{\texttt{lc-close}}, id$) to Channels (where id is the identifier of β in Σ). If there is no virtual channel open over β then in Step 2 within time Δ channel β is closed. More precisely, $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins are sent to β.Alice's account on $\mathcal{L}$, and $\beta.\mathsf{balance}(\beta.\mathsf{Bob})$ coins ares sent to β.Bob's account on $\mathcal{L}$. Then, channel β is erased from Σ, and a message ($\underline{\texttt{lc-closed}}$) is sent to the parties in β.end-users and to the adversary (sending this message to the adversary corresponds to the fact that the information that a ledger channel is closed is public).

C. The security definition

For the sake of simplicity we will consider "restricted" environments in our security definitions, i.e. we will make some explicit assumptions about $\mathcal{Z}$'s behavior. These restrictions could be eliminated at a cost of a more complex protocol construction. To save space we provide a full list of these restrictions in Appx. E. One can think about these restrictions as corresponding to assumptions about how the users use the protocol. Most of them are very natural, and they can be informally captured as "the environment never asks the honest users to do something obviously wrong", e.g., open to different channels with the same identifier, or open a channel without having sufficient funds. We also assume that if one honest party receives a message that tells her to open a channel, then all the other honest parties involved in the channel also receive it (in real life this corresponds to an assumption that they agreed via some other method, that is beyond the scope of our protocol, that the channel has to be open). We also require that P should never initiate the ledger channel closing if she earlier attempted to open virtual channel whose validity has not expired yet.

Informally speaking, a protocol `channels` is said to implement the Channels functionality, if for every environment $\mathcal{Z}$ (that satisfies the restrictions listed above) for every adversary $\mathcal{A}$ there exists an "ideal-world adversary", called a *simulator* that can produce a view for the environment $\mathcal{Z}$ that is computationally indistinguishable from $\mathcal{Z}$'s view in the real execution of the protocol `channels`. This can be formalized in a standard way (see Appx. F).

IV. The protocol and the contract

In this section we present the protocol `channels` and the contract functionality $\mathcal{C}$ that implement the Channels functionality. A formal description of the protocol appears on Figs. 1 and 2, and the contract functionality appears on Fig. 3. As already mentioned in Sect. III-A1 the contract functionality $\mathcal{C}$ maintains a set of contract *instances*. In our case every contract instance will correspond to a ledger channel. Each contract instance has a unique *identifier* (which, in our case, will be identical to the identifier of the corresponding ledger channel). A new contract instance is created when $\mathcal{C}$ receives a *constructor message* ($\underline{\texttt{lc-open}}, \beta$) (where β is a basic channel) with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins from $\beta.\mathsf{Alice}$. We refer to a "contract instance with identifier id" as $\mathcal{C}(id)$. We also say that a message m is "sent to $\mathcal{C}(id)$" or "sent by $\mathcal{C}(id)$" to denote interaction with this specific contract instance. One can also think about it in the following way: every message (other than the constructor message) that is sent to $\mathcal{C}$ contains the identifier id that specifies to which particular contract instance it is addressed, and a similar rule applies to messages sent by $\mathcal{C}$. A contract instance can also be *closed*, meaning that it terminates, and will not perform any more operations.

Each contract instance will be presented as a separate thread in $\mathcal{C}$ (and therefore the instances cannot directly refer to each other's variables). Hence each of them can be easily implemented as a separate contract on the Ethereum ledger. Most of the time the contracts will be in the *idle state*, i.e., they will wait for a function call from a party. It is important to emphasize that contracts do not take any actions by themselves, in other words: they are typically in an idle state and need to be triggered by a message from some party P_i in order to take any action. While describing the protocol we will also provide the running time of the procedures in the "normal case" (i.e. when every party involved in the procedure is honest), and in the "pessimistic case" (i.e. when some dishonest participants delay protocol's execution). Before explaining our protocol, let us start with some notation conventions.

1) "Forced reply" messages. One of the technical nuisances when dealing with the smart contracts is that they never "act by themselves", i.e., they always need a message to be woken up from an idle state. This problem appears, e.g., when a party P sends a message m to a contract to which another party P' should react within some time τ. If P' does not do it, then the contract will not automatically take any actions, and hence it needs to be "woken up" by a message from P. Since this situation appears frequently in our protocol we introduce the following convention (that can be viewed as a "macro" for writing protocols). We say that P *sends to* $\mathcal{C}(id)$ *a* Δ*-forced reply message* m if: (1) P immediately sends a message m to $\mathcal{C}(id)$ (let τ be the time when $\mathcal{C}(id)$ receives m), (2) if P does not receive a reply to m from $\mathcal{C}(id)$ within time $\tau + \Delta$ then she sends a message $(\texttt{timeout})$ to $\mathcal{C}(id)$. The "$(\texttt{timeout})$" message serves precisely the purpose of "waking the contract up". Typically, after receiving it, the contract will check if time Δ indeed passed and if P' has not replied to m'. If yes, then the contract will take appropriate actions.

2) Closing channels when cheating is detected. Sometimes the contract instance is able to detect that one of the users of channel γ is cheating (e.g. if a party does not reply to a "forced reply" message). In such cases we simply transfer all the money from γ to the party that reported such cheating. This will be handled by subroutine (C) on Fig. 3.

A. Ledger channels

1) Channel opening. We start with describing a procedure in which β.Alice and β.Bob open a ledger channel. The contract code that is responsible for creating the contract appears on Fig. 3 (A) and the protocol for the parties appears on Fig. 1 (A). Let us now explain its steps. The procedure starts when the parties receive an "lc-open" message from the environment. To open a channel β, in Step 1 (Fig. 1 (A)) party β.Alice sends to $\mathcal{C}$ a contract constructor message for $\mathcal{C}(\beta.\mathsf{id})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins. This is a "Δ-forced reply message" meaning that β.Alice sends a $(\texttt{timeout})$ message if she does not receive a reply from $\mathcal{C}(\beta.\mathsf{id})$ within time Δ after the contract instance $\mathcal{C}(\beta.\mathsf{id})$ appeared on the ledger.

In time at most Δ the contract appears on the ledger. The contract defines a transfer function *transfers* : $\beta.\mathsf{end\text{-}users} \to \mathbb{R}$ initially equal to 0 on both inputs. This function will be kept in contract's storage until the contract is closed. Its goal is to keep track on the sum of the transfers between β.Alice and β.Bob that were communicated to the contract (in our case these transfers will come only from the closing of virtual channels, see Steps 4.a and 4.c on Fig. 3 (B)). The contract will also store information about virtual channels (built on top of β) that were closed "via the contract". Technically, we will say that some channel γ is *marked as closed* if it is added to the list of such closed channels.

The contract sends a message to β.Bob informing him about the fact that β.Alice initiated ledger channel opening. If β.Bob also wants to open the channel (i.e. if he also received the "lc-open" message from the environment) then he reacts by sending a confirmation message to $\mathcal{C}(\beta.\mathsf{id})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Bob})$ coins (this happens in Step 2.a). Once the contract gets this message from β.Bob then the channel is opened (which is communicated to the parties by a message lc-opened). If this message does not arrive to the contract within time Δ then β.Alice "automatically" sends a $(\texttt{timeout})$ message, and she gets her coins back. It is easy to see that normally this procedure takes time at most 2Δ, and at most 3Δ in the pessimistic case (i.e. when β.Bob does not send his message to $\mathcal{C}$). Once a party $P \in \beta.\mathsf{end\text{-}users}$ successfully opens a ledger channel, it goes to the idle state (Step 4, Fig. 1 (A)) and waits for messages that concern β, to which he reacts, typically by going to one of the procedures described in Points (B) and (C) of the same figure.

2) Channel updating. We now describe the update procedure (it is presented formally on Fig. 1 (B)). Let us start with introducing some notation. Let $w \in \mathbb{N}$ be a natural number called a *version number*, and $\alpha \in \{0,1\}^*$ be an update annotation (see Sect. III-B). Then $(\widehat{\delta}, w, \alpha)$ is called *a version of* δ if $\widehat{\delta}$ is equal to δ on all attributes except of $\delta.\mathsf{balance}$, and the value of $\widehat{\delta}$ is equal to the value of δ. Moreover, $(\widehat{\delta}, w, \alpha, \sigma)$ is called *a version of* δ *signed by* P if $(\widehat{\delta}, w, \alpha, \sigma)$ is a tuple signed by P. If $w = 0$ then we call $(\widehat{\delta}, w, \alpha)$ the *initial* version of δ, and in the "signed" tuple $(\delta, w, \alpha, \sigma)$ we allow $\sigma = \bot$.

The channel updates will be done by exchanging signatures on subsequent versions of δ, i.e., the w-th update will have a version number w, The *winner selection procedure* Win serves to determine which version of a channel is newer. It is defined as follows. Let δ be a ledger or virtual channel. Win takes as input a pair $((\delta^0, w^0, \alpha^0, \sigma^0), (\delta^1, w^1, \alpha^1, \sigma^1))$ of signed versions of δ, and returns as output a cash function $\theta : \delta.\mathsf{end\text{-}users} \to \mathbb{R}_{\geq 0}$ defined as follows: let i be such that $w^i > w^{1-i}$ (if no such i exists then choose $i := 0$) and then let $\theta := \delta^i.\mathsf{balance}$.

To present the main idea behind the channel updates, we first describe the update procedure in the *non-parallel* setting, i.e., in the situation when only one update of a given channel δ is performed at a time (the full parallel version, presented on Fig. 1 (B) is described later in this section). The parties that opened a channel maintain the counter w denoting the

(A) **Opening ledger channel** β:

1) Upon receiving a message $(\underline{\texttt{lc-open}}, \beta)$ from the environment, party $\beta.\mathsf{Alice}$ starts a new thread, sends to $\mathcal{C}$ a Δ-forced reply contract constructor message $(\texttt{lc-open}, \beta)$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins, and goes to Step 3.

2) Upon receiving a message $(\underline{\texttt{lc-open}}, \beta)$ from the environment, party $\beta.\mathsf{Bob}$ starts a new thread and waits time at most Δ to receive a message $(\texttt{lc-opening}, \beta)$ from $\mathcal{C}(\beta.\mathsf{id})$. Consider the following cases.

a) He receives this message — then he replies to $\mathcal{C}(\beta.\mathsf{id})$ with a message $(\texttt{lc-open})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Bob})$. He then waits to receive a message $(\texttt{lc-opened})$ from $\mathcal{C}(\beta.\mathsf{id})$. Once received, he outputs $(\underline{\texttt{lc-opened}})$ and goes to Step 4.

b) He does not receive this message — then he outputs $(\underline{\texttt{lc-not-opened}})$ and stops this thread.

3) Party $\beta.\mathsf{Alice}$ waits for one of the following:

a) She receives a message $(\texttt{lc-not-opened})$ from $\mathcal{C}(\beta.\mathsf{id})$ — then she outputs $(\underline{\texttt{lc-not-opened}})$ and stops this thread.

b) She receives a message $(\texttt{lc-opened})$ from $\mathcal{C}(\beta.\mathsf{id})$ — then she outputs $(\underline{\texttt{lc-opened}})$ and goes to Step 4.

4) Party $P \in \beta.\mathsf{end\text{-}users}$ goes in the idle state waiting for messages that concern β and come from the environment, $\beta.\mathsf{other\text{-}party}(P)$ or from the contract. These messages are handled in Points (B) and (C) below.

(B) **Update of (ledger or virtual) channel** δ:

1) Upon receiving message $(\underline{\texttt{update}}, id, \theta, \alpha)$ (where id is an identifier of some channel δ) from the environment party P (the "initiator") waits for the next P's update round of δ. When this round comes P lets $(\widehat{\delta}^P, w^P)$ denote the last version of δ that P is aware of, and lets $\widetilde{\delta}$ be equal to $\widehat{\delta}^P$ except that $\widetilde{\delta}.\mathsf{balance} := \widehat{\delta}^P.\mathsf{balance} + \theta$. Then she sends a tuple $(\texttt{updating}, (\widetilde{\delta}, w^P + 1, \alpha, \sigma))$ (where σ is P's signature on $(\widetilde{\delta}, w^P + 1, \alpha)$) to P', waits 1 round, and goes to Step 3

2) This step starts when $P' \in \delta.\mathsf{end\text{-}users}$ (the "confirmer") receives a correctly signed message $(\texttt{updating}, (\widetilde{\delta}, w, \alpha, \sigma))$. Then P' lets $(\widehat{\delta}^{P'}, w^{P'})$ denote the last version of δ that P' is aware of. If $w \neq w^{P'} + 1$ then P' ignores this message. Otherwise P' computes $\theta' := \widetilde{\delta}.\mathsf{balance} - \widehat{\delta}^{P'}.\mathsf{balance}$ and outputs $(\underline{\texttt{update-requested}}, \beta.\mathsf{id}, \theta', \alpha)$ to the environment. If the environment replies with $(\underline{\texttt{update-ok}})$ then P' computes her signature σ' on $(\widetilde{\delta}, w, \alpha)$, sends a message $(\texttt{update-ok}, \sigma')$ to P' and stops this procedure. Otherwise P' stops this procedure.

3) If P receives a message $(\texttt{update-ok}, \sigma')$ where σ' is a signature of P' on $(\widetilde{\delta}^P, w^P + 1, \alpha)$ then she outputs $(\underline{\texttt{updated}})$ and stops this procedure. Otherwise P outputs $(\underline{\texttt{not-updated}})$ and stops this procedure.

(C) **Closing the ledger channel with identifier** id:

1) Upon receiving a message $(\underline{\texttt{lc-close}}, id)$ (where id is an identifier of some ledger channel β) from the environment party P lets $(\beta, 0)$ be the initial version of the channel with identifier id.
Party P lets V be the last signed version of β which P received from $\beta.\mathsf{other\text{-}party}(P)$ (if P has never received such a version then she lets $V = (\beta, 0, \varepsilon, \bot)$). She sends to $\mathcal{C}(id)$ a Δ-forced reply message $(\texttt{lc-close}, V)$.

2) Upon receiving (in some round τ) a message $(\texttt{lc-closing})$ from $\mathcal{C}(\beta.\mathsf{id})$ party P' does the following:

a) If earlier she received an opening certificate z of $P := \beta.\mathsf{other\text{-}party}(P)$ on some virtual channel γ that is constructed over virtual channel β and $\gamma.\mathsf{validity} + 5\Delta + 1 > \tau$ — then she sends to $\mathcal{C}(\beta.\mathsf{id})$ a message $(\texttt{vc-active}, z)$. Then she continues waiting.

b) Otherwise she sends to $\mathcal{C}(\beta.\mathsf{id})$ a message $(\texttt{lc-close}, V')$, where V' is the last signed version of β that she received from P (if she have never received such a version then she lets $V' = (\beta, 0, \varepsilon, \bot)$). Upon receiving a message $(\texttt{lc-closed})$ from $\mathcal{C}(\beta.\mathsf{id})$ she outputs $(\underline{\texttt{lc-closed}})$ and stops this thread.

3) Upon receiving a message $(\texttt{lc-closed})$ from $\mathcal{C}(\beta.\mathsf{id})$ party P outputs $(\underline{\texttt{lc-closed}})$ and stops this thread.

Fig. 1: The procedures: (A) "opening ledger channel", (B) "update of channel δ", and (C) "closing the ledger channel" (see Sect. IV-A.

channel version. Initially w is set to 0, and it is incremented after each update of δ. Let P be the initiator of the update, and let P' be the confirmer (see Sect. III-B). Suppose that for each each $\widehat{P} \in \{P, P'\}$ the last version of δ that $\widehat{P}$ is aware of is $W^{\widehat{P}} := (\widehat{\delta}^{\widehat{P}}, w^{\widehat{P}}, \alpha^{\widehat{P}})$ (with $W^{\widehat{P}} := (\delta, 0, \varepsilon)$ if no update has been performed yet). Of course, if both parties are honest, then $W^P = W^{P'}$. An update that performs a transfer θ on channel δ, is annotated with a string α and is initiated by P (see Sect. III-B) works as follows:

Proposing an update. Let $\widetilde{\delta}$ be defined as $\widehat{\delta}^P$ except that $\widetilde{\delta}.\mathsf{balance} := \widehat{\delta}^P.\mathsf{balance} + \theta$. Party P sends to P' a message $m = (\mathtt{updating}, (\widetilde{\delta}, w^P + 1, \alpha, \sigma))$, where σ is her signature on $(\widetilde{\delta}, w^P + 1, \alpha)$ (see Step. 1, Fig. 1 (B)).

Accepting an update. Upon receiving m party P' checks if the signature and the version number are correct. If yes, then she has to decide if this update is ok. To this end she first computes what transfer has been proposed by P. This is done by calculating θ' as $\theta' := \widetilde{\delta}.\mathsf{balance} - \widehat{\delta}^{P'}.\mathsf{balance}$ (in other words: P' recovers the transfer function used by P to calculate from m). Then P' outputs a "update-requested" message to the environment. If the environment replies with "update-ok" then P' confirms the update by sending her on $(\widetilde{\delta}, w^P + 1, \alpha)$ back to P. The details appear in Step 1 of Fig. 1 (B).

Let us now discuss the full parallel settings. In reality the update procedure is more complicated than what we described above, since it may happen that more than one channel update request is issued in the same round. One of the issues that needs to be handled is that it may happen that some updates are initiated by $\delta.\mathsf{Alice}$ and some by $\delta.\mathsf{Bob}$ in the same round (note that in this case they would use the same value of w for two different updates). To avoid this problem we define *P's update rounds for δ* as the rounds when P can send the first message in the "initiating an update" phase (if the update procedure has been called in some other round then P waits for his update round to start). More precisely, for a channel δ round τ is called a $\delta.\mathsf{Alice}$*'s update round* if $\tau = 0 \pmod 4$ and it is called a $\delta.\mathsf{Bob}$*'s update round* if $\tau = 2 \pmod 4$. Since a channel update takes 2 rounds, therefore having only 1 in 4 rounds as an "update round" for P guarantees that the entire update procedure proposed by P will end before P' starts any procedure containing her update proposal. Note also that, since we assumed that the adversary cannot reorder messages sent from P to P', the version number w will remain synchronized between the parties.

Observe that as a result of a successful update procedure each party has a signature of the other party on a tuple containing a string α (i.e. a tuple "$(\widetilde{\delta}, w^P + 1, \alpha, \sigma)$"), that can be later used to prove that a given transfer indeed happen. We will use it in the virtual channel closing protocol where string α will contain an information about the virtual channel that has been closed (this will be called the "closing certificate" see, e.g., Fig. 2, (B), Step 3). A successful execution of the update procedure takes at most 5 rounds (since a party may need to wait at most 3 rounds for her "update round", and then 2 rounds for a reply from the other party). If the update procedure finishes in 5 rounds then we say that it was *successful*, otherwise we say it *failed*.

3) Channel closing. At some point one of the parties, let us say P, gets a request from the environment to close a ledger channel β (let P' be the other party in $\delta.\mathsf{end\text{-}users}$). The formal description of the closing procedure appears on Fig. 1 (C), and the corresponding part of the contract is described in Step 4 of Fig. 3 (B). We now explain it informally (for a moment we ignore Step 2.a on Fig. 1 (C), and Step (4.b) on Fig. 3 (B) — we will explain them a bit later, when we talk about virtual channel opening).

Initially (see Step 1, Fig. 1 (C)) Party P sends to the contract $\mathcal{C}(\beta.\mathsf{id})$ a Δ-forced reply message containing the latest signed version V of β that she received from P'. Of course the contract has no reason to believe P that this indeed is the latest version of β. This is why the contract sends an "lc-closing" message to P' informing her about the fact that P initiated channel closing (see Step 4, Fig. 3 (B)). Party P' then replies with her most recent version P' of β (see Step (b), Fig. 1 (C)). The contract then applies the Win function to both V and V' to determine the value of the balance function that should be used for channel closing. He also "corrects" this value by taking into account the values of transfers described by the function *transfers*. Altogether, he lets $\mathit{balance} := \mathsf{Win}(V, V') + \mathit{transfers}$. He then distributes the coins between P and P' according to $\mathit{balance}$, and stops this contract instance (see Step 4.a, Fig. 3 (B) for details). Of course, it may happen that P' does not reply to the lc-closing message. In this case P sends a (timeout) message to the contract (this happens automatically, since the first message from P is "Δ-forced reply). The contract reacts by assuming that P's version of the channel is the valid one (see Step 4.c, Fig. 3 (B)). It is easy to see that normally this procedure takes time at most 2Δ, and at most 3Δ in the pessimistic case (i.e. when P' does not send her message to $\mathcal{C}$).

B. Virtual channels

We now describe the protocols for virtual channel opening and closing (the protocol for virtual channel *update* has already been described in Sect. IV-A2). Let γ be a virtual channel, and suppose that there exist the following open ledger channels: $\beta_{\gamma.\mathsf{Alice}}$ between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$ and $\beta_{\gamma.\mathsf{Bob}}$ between $\gamma.\mathsf{Bob}$ and $\gamma.\mathsf{Ingrid}$ with $\beta_P.\mathsf{id} = \gamma.\mathsf{subchan}(P)$ for $P \in \{\beta.\mathsf{Alice}, \beta.\mathsf{Bob}\}$. As already explained in Sect. II, a virtual channel γ is built with the help of $\gamma.\mathsf{Ingrid}$ using ledger channels $\beta_{\gamma.\mathsf{Alice}}$ and $\beta_{\gamma.\mathsf{Bob}}$ in a similar way as the ledger channels are built with the help of the ledger. Unfortunately, there are some important differences between the functionality that the ledger channels and the ledger provide.

Firstly, the ledger channels described in Sect. IV-A serve only the purpose of performing payments between two parties, and they do not allow to execute smart contracts "inside of the channel" (while the ledger allows it). We solve this problem using the concept of "state channels" [1]. To keep the paper concise we do not introduce this type of channels formally here (this is done in future work, see Appx. C). Essentially, the

idea of the state channels is to enrich the payment channels with additional functionality that makes them behave like a "2-party ledger with contracts". This is done by allowing the parties exchange signatures on some additional strings that can later be interpreted by the channel contract. In our case, these strings will called be the "opening" and "closing" certificates. Our channel contract will interpret these signed strings, and in fact its major part serves only this purpose (see Points 1— 3, and Point 4.b on Fig. 3 (B)). The second problem is that the state channels give us a "virtual ledger" for only 2 parties. In other words: what happens in a channel β_P is "invisible" for $\gamma.\mathsf{other\text{-}party}(P)$ (which is different from the global ledger that is used to build ledger channels). This results in slightly more complicated protocols. We now describe our solution in more detail.

1) Channel opening. The virtual channel opening protocol is presented on Fig. 2 (A). The parties decide to open the channel γ once they receive a "vc-opened" message from the environment. For $P \in \gamma.\mathsf{all\text{-}users}$ an *opening certificate of* P *on* γ is a pair $z_P := (\mathtt{opening}, \gamma, \sigma_P)$, where σ_P is P's signature on γ. The role of these certificates is to guarantee that a party $P \in \gamma.\mathsf{all\text{-}users}$ cannot deny that she agreed to open γ (when the parties interact with the contract). Let us first describe the process of opening a virtual channel in case all parties are honest. In Step 1 both parties $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ send their opening certificates on γ to $\gamma.\mathsf{Ingrid}$. If $\gamma.\mathsf{Ingrid}$ receives *both* of these certificates in the next round, then she replies to $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ with her opening certificate on γ (this happens in Step 2.a), and considers the channel open. Parties $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ receive this certificate in the next round (Step 3), forward this certificate to each other (we will explain in a moment the role of this forwarding), and also consider the channel open. Note that technically virtual channel opening does not result in removing coins from parties' account in the ledger channels (they will be removed later, when the virtual channel is closed, see Step 3, Fig. 2 (B)). Such "delayed removal" is ok, since the parties locally can anyway keep track on how many coins are still not blocked in their ledger channels. Once a party considers a channel opened then she outputs a message vc-opened and goes to an idle state (see Step 4) where she waits for channel update request (that are handled by the procedure already described in Sect. IV-A2), or for time $\gamma.\mathsf{validity}$ to come.

Now consider what happens if some of the parties are misbehaving. In this case the execution of the protocol can result in *not* opening channel γ. Informally, the main properties that our protocol needs to have are: (1) if some parties cheat, then no honest party looses coins, and (2) there is a consensus between the honest parties on whether the channel has been open or not. Let us first show how our protocol provides security against (1). The result of the protocol execution is that the parties end up with knowledge of opening certificates of some of the other parties on γ. Since such a certificate can later be used to claim coins from a party P that signed it, thus the main security risk for P is that she signs a certificate that will later be used to claim coins from P, while P cannot claim coins from other parties in $\gamma.\mathsf{all\text{-}users}$ since she herself did not receive an opening certificate on γ. It is easy to see that this problem does not occur for $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$. This is because these parties will not consider the channel open if they do not receive an opening certificate from $\gamma.\mathsf{Ingrid}$, and in this case they will never perform any update to γ. Therefore even if a malicious $\gamma.\mathsf{Ingrid}$ does not send an opening certificate on γ to $\gamma.\mathsf{end\text{-}users}$, and then requests to close γ (when time $\gamma.\mathsf{validity}$ comes, see Sect. IV-B2 below), the result of her behavior will be "neutral" for both $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ (as the "default" state of γ is that both parties get the same amount of coins as they deposited). Therefore what remains is to show that (1) is satisfied for $\gamma.\mathsf{Ingrid}$. Here, the problem could potentially be larger, as $\gamma.\mathsf{Ingrid}$ could loose coins if she sends her certificate to one party $P \in \gamma.\mathsf{end\text{-}users}$ without getting the certificate from $P' = \gamma.\mathsf{other\text{-}party}(P)$ (as during the channel closing she would be forced to pay coins to P *without* being guaranteed that she get the same amount of coins from P'). This problem is precisely the reason why in our protocol $\gamma.\mathsf{Ingrid}$ signs the opening certificates only if she received the opening certificates on γ from *both* $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$. In other words: she only commits herself to cover P's commitments in front of P' if she is guaranteed that P can be held responsible for these commitments.

Now let us discuss (2). In the security proof (see Appx. G) we will consider γ "open" if at least one of $\gamma.\mathsf{end\text{-}users}$ received an opening certificate from $\gamma.\mathsf{Ingrid}$ in Step 3. Hence, the only situation when there is disagreement between the *honest* parties (on whether γ has been open or not) is if both $\gamma.\mathsf{end\text{-}users}$ are honest, and a malicious $\gamma.\mathsf{Ingrid}$ sends her opening certificate to some $P \in \gamma.\mathsf{end\text{-}users}$, and does *not* send it to $P' := \gamma.\mathsf{other\text{-}party}(P)$. To avoid this problem we let the parties in Step 3 forward to each other the opening certificate from $\gamma.\mathsf{Ingrid}$. This guarantees that if at least one of $\gamma.\mathsf{end\text{-}users}$ considers the channel open, then the other one considers it open to. It is easy to see that normally opening a channel takes 2 rounds. In the pessimistic case (when $\gamma.\mathsf{Ingrid}$ sends her opening certificate to one party only) it takes 3 rounds.

Finally, let us comment on the behavior of the parties when the opening procedure ends. First consider a successful opening. One thing that would obviously be illegal is if a party from $\gamma.\mathsf{all\text{-}users}$ starts the ledger channel closing procedure (see Fig. 1 (C)) for one of the ledger channels in $\gamma.\mathsf{subchan}$ before time $\gamma.\mathsf{validity} + 5\Delta + 2$ comes (the "$5\Delta + 2$" term comes from the fact that, as we show below, closing a virtual channel can take time at most $5\Delta + 2$, and during this time the ledger channel should still be available). Therefore after every successful opening of a virtual channel γ, each party $P' \in \gamma.\mathsf{all\text{-}users}$ monitors the situation in the ledger channel β_P and reacts to the requests of $P := \beta.\mathsf{other\text{-}party}(P')$ to close it. Recall that if a closing request was made by P (and passed to P' via the contract, see Step 4 on Fig. 3 (B)) then it is handled by the closing procedure on Fig. 1 (C). In Step 2.a of the closing procedure the party P' reacts by sending a message $(\mathtt{vc\text{-}active}, z)$ to the contract (where z is the opening

(A) **Opening virtual channel γ**:

1) Upon receiving a message $(\texttt{vc-open}, \gamma)$ from the environment each party $P \in \gamma.\mathsf{end\text{-}users}$ starts a new thread, and sends to $\gamma.\mathsf{Ingrid}$ her opening certificate on γ, waits one round and goes to Step 3
2) Upon receiving a message $(\texttt{vc-open}, \gamma)$ from the environment party $\gamma.\mathsf{Ingrid}$ waits one round to receive opening certificates on γ of both $P \in \gamma.\mathsf{end\text{-}users}$. Consider the following cases.
   a) She receives both opening certificates: then she replies to each $P \in \gamma.\mathsf{end\text{-}users}$ with her opening certificate on γ. Then she outputs $(\texttt{vc-opened})$, waits until round $\gamma.\mathsf{validity}$ and then goes to the "closing virtual channel procedure".
   b) Otherwise: she outputs $(\texttt{vc-not-opened})$ and stops.
3) If a party $P \in \gamma.\mathsf{end\text{-}users}$ receives an opening certificate of $\gamma.\mathsf{Ingrid}$ on γ from $\gamma.\mathsf{Ingrid}$ then she forwards this certificate to $\gamma.\mathsf{other\text{-}party}(P)$, outputs $(\texttt{vc-opened})$ and goes to Step 4 below.
   Otherwise $P \in \gamma.\mathsf{end\text{-}users}$ waits one more round to receive an opening certificate of $\gamma.\mathsf{Ingrid}$ on γ from $\gamma.\mathsf{other\text{-}party}(P)$. If she receives it then she outputs $(\texttt{vc-opened})$ and goes to goes to Step 4 below.
   Otherwise she outputs $(\texttt{vc-not-opened})$, waits for time $\gamma.\mathsf{validity}$ and then goes to the closing procedure in Point (B) below.
4) Party $P \in \gamma.\mathsf{all\text{-}users}$ goes in the idle state waiting for the "channel update" messages that concern γ and come from the environment or $\gamma.\mathsf{other\text{-}party}(P)$ or from the contract. These messages are handled by the procedure described of Fig. 1 (B). When time $\gamma.\mathsf{validity}$ comes P goes to Point (B) below.

(B) **Virtual channel closing**:

For $P \in \gamma.\mathsf{end\text{-}users}$ let β_P denote the channel with identifier $\gamma.\mathsf{subchan}(P)$, and let $(\gamma_0, 0)$ be the initial version of channel γ. For $P \in \gamma.\mathsf{all\text{-}users}$ let z_P denote the opening certificate of P on γ.

1) In round $\gamma.\mathsf{validity}$ each $P \in \gamma.\mathsf{end\text{-}users}$ lets $W^P := (\gamma^P, v^P, \alpha^P, \sigma^P)$ be the latest signed version of γ that P received from $P' := \gamma.\mathsf{other\text{-}party}(P)$. If P never received a signed version of γ from P' (which means that no updates of γ have been performed) then P lets $W^P := (\gamma_0, 0, \varepsilon, \bot)$. Then P sends to $\gamma.\mathsf{Ingrid}$ a tuple $(\texttt{vc-close}, W^P, \mathsf{Sign}_P(W^P))$ and goes to Step 4.
2) In round $\gamma.\mathsf{validity} + 1$ party $\gamma.\mathsf{Ingrid}$ does the following for each $P \in \gamma.\mathsf{end\text{-}users}$:
   a) If she receives a correctly formated $(\texttt{vc-close}, W^P, S^P)$ message from P then she goes to Step 3.
   b) Otherwise she sends a Δ-forced reply message $(\texttt{vc-close}, z_P)$ to $\mathcal{C}(\beta_P.\mathsf{id})$. If she then receives a message $(\texttt{vc-close-init}, W^{P'}, S^{P'})$ from $\mathcal{C}(\beta_P.\mathsf{id})$ then goes to Step 3. Otherwise she receives a message $((\texttt{vc-closed})$ — in this case she sets $W^P := (\gamma_0, 0, \varepsilon, \bot)$ and $S_P := \bot$ and goes to Step 3.
3) Party $\gamma.\mathsf{Ingrid}$ waits to learn (W^P, S^P) for *both* $P \in \gamma.\mathsf{end\text{-}users}$ (either by getting (W^P, S^P) directly from P, or "via the contract").
   She then lets $\theta := \mathsf{Win}(W^{\gamma.\mathsf{Alice}}, W^{\gamma.\mathsf{Bob}})$. Then for each $P \in \gamma.\mathsf{end\text{-}users}$ (such that such that channel β_P has not been closed) she proposes an update of β_P that adds $x := \theta(P) - \gamma_0.\mathsf{balance}(P)$ coins to P's account and $-x$ coins to $\gamma.\mathsf{Ingrid}$'s account and is annotated with a string "$\texttt{channel}\ \gamma.\mathsf{id}\ \texttt{closed}$", and goes to Step 5.
4) Party $P \in \gamma.\mathsf{end\text{-}users}$ waits for one of the following events to happen:
   a) Party $\gamma.\mathsf{Ingrid}$ proposes an update to ledger channel β_P that adds at least $\gamma^P.\mathsf{balance}(P) - \gamma_0.\mathsf{balance}(P)$ coins to P's account and is annotated with a string "$\texttt{channel}\ \gamma.\mathsf{id}\ \texttt{closed}$" — then P confirms this update, outputs $(\texttt{vc-closed})$ and goes to Step 6.
   b) Party P receives a message $(\texttt{vc-closing}, \gamma.\mathsf{id})$ from $\mathcal{C}(\beta_P.\mathsf{id})$ then P replies with $(\texttt{vc-closing}, W^P, \mathsf{Sign}_P(W^P))$ and continues waiting
   c) Within round $\gamma.\mathsf{value} + 2\Delta + 1$ none of the above happens then P sends a Δ-forced reply message $(\texttt{vc-close-timeout}, z_{\gamma.\mathsf{Ingrid}})$ to $\mathcal{C}(\beta_P.\mathsf{id})$, outputs $(\texttt{vc-closed})$ and stops.
5) If the update procedure is successful then $\gamma.\mathsf{Ingrid}$ outputs $(\texttt{vc-closed})$ and goes to Step 6.
   Otherwise she sends to $\mathcal{C}(\beta_P.\mathsf{id})$ a Δ-forced reply message $(\texttt{vc-close-final}, z_P, (W^{\gamma.\mathsf{Alice}}, S^{\gamma.\mathsf{Alice}}), (W^{\gamma.\mathsf{Bob}}, S^{\gamma.\mathsf{Bob}}))$. Once she receives a message $(\texttt{vc-closed})$ from $\mathcal{C}(\beta_P.\mathsf{id})$ she outputs $(\texttt{vc-closed})$ and stops this procedure.
6) A party $P \in \gamma.\mathsf{all\text{-}users}$ goes in to an idle state. If at any point later P receives a message from $\mathcal{C}$ that concerns channel γ then P answers with $(\texttt{vc-already-closed}, z)$, where z is the closing certificate on γ (see Sect. IV-B2).

Fig. 2: The procedures: (A) "opening virtual channel" (see Sect. IV-B1) and (B) "virtual channel closing" (see Sect. IV-B2).

certificate). After receiving this message the in Step 4.b (Fig. 3 (B)) the contract decides that the party P is corrupt, and hence it closes, and gives all its coins to P'.

A subtle attack that a dishonest γ.Ingrid could execute is: (a) after receiving P's opening certificate on γ do not send your opening certificate on γ to P (in Step 2.a), so that P concludes that the channel γ has not been open, and then (b) if P requests to close the underlying ledger channel β in time earlier than γ.validity then send a message $(\texttt{vc-active}, z)$ to the contract (i.e.: claim that P is dishonest). To prevent this from happening we have a requirement that P should never initiate the ledger channel closing if she earlier attempted to open virtual channel whose validity has not expired yet (see Sect. III-C).

2) Channel closing. The virtual channel closing procedure is depicted on Fig. 2 (B). It is started automatically when time γ.validity comes. The main idea of this procedure is that it is γ.Ingrid who is responsible for closing γ and taking care that the channels $\beta_{\gamma.\mathsf{Alice}}$ and $\beta_{\gamma.\mathsf{Bob}}$ are updated in the correct way (i.e. according the the latest balance of γ). If everybody is honest then the procedure works in a pretty straightforward way. Let us now explain it, ignoring some details that are need to prevent cheating from dishonest parties. Let $(\gamma_0, 0)$ denote the initial version of channel γ.

First, in Step 1 each party $P \in \gamma$.end-users sends to γ.Ingrid the latest signed version W^P of γ that she received from the other party (or the initial version of γ if no such update has been performed). Then in Step 3 party γ.Ingrid decides what is the latest balance of γ by checking which version has a higher number (this is done using the Win procedure) and proposes to update the ledger channels accordingly. Finally, in Step 4.a the parties in γ.end-users confirm the update, and channel γ is closed. The ledger channel update is annotated with a string $\xi =$"channel γ.id closed". Hence a successful update procedure of each β_P results in every $\widehat{P} \in \beta_P$.end-users holding a signature $\widehat{\sigma}$ of β_P.other-party$(\widehat{P})$ on a tuple W that includes string ξ. Call a pair $z = (W, \widehat{\sigma})$ a *closing certificate of* β_P.other-party$(\widehat{P})$ *on* γ. The role of the closing certificates is to prevent a malicious party from requesting a channel virtual channel closing "via the contract" after the channel has already been closed by mutual agreement. If at some point a request like this is issued by a party $\widehat{P}' := \beta_P$.other-party$(\widehat{P})$ to a contract $\mathcal{C}(\beta_P.\mathsf{id})$ then $\widehat{P}$ replies with a message $(\texttt{vc-already-closed}, z)$ (see Step 6, Fig. 2 (B)), and the contract decides that a $\widehat{P}'$ is cheating (see Steps 1.a and 2.a, Fig. 3 (B)). There are several other things that need to be taken care of to make the closing protocol secure against dishonest behavior of some parties. Note that the contract functionality $\mathcal{C}$ is not aware of the virtual channel until a disagreement happens between the parties. Therefore the parties need a way to prove to $\mathcal{C}$ that a channel has indeed been open. This is done by sending the opening certificates to it. For $P \in \gamma$.all-users let z_P denote the opening certificate of P on γ. Firstly, observe that a malicious $P \in \gamma$.end-users can send an old version of the channel to γ.Ingrid, but this will not count since if the other party $P' := \gamma$.other-party(P) is honest, then the version submitted by P will "loose" against P's version. One subtle problem could appear if P is honest, P' is dishonest, and P' earlier did not confirm an update of γ that was initiated by P. In this case P' can submit higher version signed of γ than P (since P did not receive P''s signature on this version). This is solved by letting each P accept an update that gives her *at least* the amount of coins that she expected (see Point (4.a), Fig. 2 (B)). Note that by the restriction on the environment that we made in Sect. III-C the newer "unconfirmed" version of γ can only be such that P gets *more* coins than before the update.

Other obvious type of malicious behavior of by the parties is: not sending the messages that a party was supposed to send in a given round. If γ.Alice or γ.Bob do so, then γ.Ingrid talks to them "via the contract" to extract the necessary information. This is done by sending a "forced reply" message $(\texttt{vc-close-init}, z_P)$, (in Step 2.b, Fig. 2 (B)) to a contract, which is received by it in Step 1 (see Fig. 3 (B)). A contract sends a $(\texttt{vc-closing}, \gamma.\mathsf{id})$ message to P, which is handled by P in Step 4.b (Fig. 2 (B)). If P remains silent then the contract instance that handles the given ledger channel gets closed (thanks to the "forced reply" mechanism) and all its coins are given to γ.Ingrid.

If some of the ledger channel updates fails then in Step (5) (Fig. 2 (B)) γ.Ingrid asks the contract to handle the channel closing by sending to it both signed versions of γ. The contract handles this request in Step 2, and (if the other party does not signal that virtual channel γ has already been closed) contract stores in its memory the result of the virtual channel closing. Technically, this is done by adjusting the value of a function *transfers* (in Step 2.b, Fig. 3 (B)) to take into account the transfers resulting from closing the virtual channel. Note that an honest γ.Ingrid will normally close the virtual channel in at most 6 rounds (one round is needed to obtain the W^P's and at most 5 rounds to finish the update of β_P's). Pessimistically (i.e. when γ.end-users delay the execution) γ.Ingrid learns both (W^P, S^P)'s in Step 3 in time at most γ.validity $+ 2\Delta + 1$. This is because in the worst case γ.Ingrid has to send the timeout message to the contract in time at least $\Delta + 1$, (see Step 1.c, Fig. 3), and the reaction of the contract can take another Δ rounds. If this does not happen then in the next round (i.e. γ.validity $+ 2\Delta + 1$) each party $P \in \gamma$.end-users sends (in Step 4.c, Fig. 2 (B)) a Δ-forced reply message $(\texttt{vc-close-timeout}, z_{\gamma.\mathsf{Ingrid}})$ to the contract. The contract receives it (in Step 3) after at most Δ rounds, i.e. in round at most γ.validity $+ 3\Delta + 1$, and check if γ indeed has not been closed. This is done by sending a message $(\texttt{vc-closing}, \gamma.\mathsf{id})$ to γ.Ingrid. If γ.Ingrid proves that γ has been closed (by sending a closing certificate on γ to the contract) then the contract concludes that P is cheating (see Sect. IV-2) and closes this contract instance. Otherwise, after time Δ passes P sends to the contract a $(\texttt{timeout})$ message and the contract concludes that γ.Ingrid is cheating. The contract instance is then closed in time at most γ.value $+ 5\Delta + 1$. Hence the entire closing procedure takes time at most $5\Delta + 1$ in the pessimistic case.

(A) **The contract for channel β opening**:

Upon receiving a contract constructor message $(\texttt{lc-open}, \beta)$ with $\beta.\mathsf{balance}(\beta.\mathsf{Alice}))$ coins from $\beta.\mathsf{Alice}$:
Start a new contract instance with identifier $\beta.\mathsf{id}$. Send a message $(\texttt{lc-opening}, \beta)$ to $\beta.\mathsf{Bob}$ and wait for one of the following to happen:

1) You receive a message $(\texttt{lc-open})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Bob})$ coins from $\beta.\mathsf{Bob}$:
Let *transfers* : $\beta.\mathsf{end\text{-}users} \to \mathbb{R}$ be a transfer function initially equal to 0 on both inputs. Send a message $(\texttt{lc-opened})$ to $\beta.\mathsf{end\text{-}users}$ and go to "Contract execution" below.
2) You receive a message $(\texttt{timeout})$ from $\beta.\mathsf{Alice}$ in time at least Δ after you sent the message to $\beta.\mathsf{Bob}$: close the contract and send a message $(\texttt{lc-not-opened})$ together with $\beta.\mathsf{balance}(\beta.\mathsf{Alice})$ coins to $\beta.\mathsf{Alice}$.

(B) **The contract $\mathcal{C}(id)$ execution**:

Assumption: for every channel δ each party P can send at most one message of a given type that concerns δ.
Wait for messages from parties $\beta.\mathsf{end\text{-}users}$. Consider the following cases.

1) You receive a message $(\texttt{vc-close-init}, (\gamma, \sigma))$ from $\gamma.\mathsf{Ingrid}$ in time at least $\gamma.\mathsf{validity} + 2$ (where (γ, σ) is an opening certificate of $P := \beta.\mathsf{other\text{-}party}(\gamma.\mathsf{Ingrid})$ on γ) and γ has not been marked as closed — then send a message $(\texttt{vc-close-init}, \gamma.\mathsf{id})$ to P and wait for one of the following:
   a) You receive a message $(\texttt{vc-already-closed}, z)$ from P, where z is closing certificate of $\beta.\mathsf{other\text{-}party}(P)$ on γ — in this case mark γ as closed.
   b) You receive a message $m := (\texttt{vc-close}, W, \mathsf{Sign}_P(W))$ from P (where W is a version of γ signed by $\gamma.\mathsf{other\text{-}party}(P)$) — then send m to $\gamma.\mathsf{Ingrid}$.
   c) You receive a message $(\texttt{timeout})$ from $\gamma.\mathsf{Ingrid}$ in time at least Δ after you sent the message $(\texttt{vc-close-init}, \gamma.\mathsf{id})$ — in this case go to subroutine (C) below.
2) You receive a message $m := (\texttt{vc-close-final}, (\gamma, \sigma), (W^{\gamma.\mathsf{Alice}}, S^{\gamma.\mathsf{Alice}}), (W^{\gamma.\mathsf{Bob}}, S^{\gamma.\mathsf{Bob}}))$ from $\gamma.\mathsf{Ingrid}$ where (γ, σ) is an opening certificate of $P := \beta.\mathsf{other\text{-}party}(\gamma.\mathsf{Ingrid})$ on γ, each W^P is a version of γ signed by $\gamma.\mathsf{other\text{-}party}(P)$, and S_P is a signature of P on W (or is equal to $\perp$ if W is the initial version of γ), and γ has not been marked as closed — send message m to P and wait for one of the following to happen:
   a) You receive a message $(\texttt{vc-already-closed}, z)$ from P, where z is a signed tuple containing a string "$\texttt{channel}\ \gamma.\mathsf{id}\ \texttt{closed}$" — in this case do nothing.
   b) You receive a message $(\texttt{timeout})$ from $\gamma.\mathsf{Ingrid}$ in time $\Delta + 1$ after you sent m to P — in this case go to subroutine (C) below.
3) You receive a message $(\texttt{vc-close-timeout}, (\gamma, \sigma))$ from $P \in \gamma.\mathsf{end\text{-}users}$ in time at least $\gamma.\mathsf{validity} + 2\Delta + 2$ where (γ, σ) is an opening certificate of $\gamma.\mathsf{Ingrid}$ on γ and γ has not been marked as closed) — send a message $(\texttt{vc-closing}, \gamma.\mathsf{id})$ to $\gamma.\mathsf{Ingrid}$ and wait for one of the following to happen:
   a) You receive a message $(\texttt{vc-already-closed}, z)$ from $\gamma.\mathsf{Ingrid}$, where z is a closing certificate of $\beta.\mathsf{other\text{-}party}(P)$ on γ — in this case do nothing.
   b) You receive a message $(\texttt{timeout})$ from P in time at least $\Delta+1$ after you sent the $(\texttt{vc-closing}, \gamma.\mathsf{id})$ message to $\gamma.\mathsf{Ingrid}$ — in this case go to subroutine (C) below.
4) You receive a message $(\texttt{lc-close}, V)$ from P, where $V = (\gamma^P, v^P, \varepsilon, \sigma)$ is a version of β signed by $P' = \beta.\mathsf{other\text{-}party}(P)$ — send a message $(\texttt{lc-closing})$ to P' and wait for one of the following to happen:
   a) P' replies with $(\texttt{lc-close}, V')$ where V' is a version of β signed by $P' = \beta.\mathsf{other\text{-}party}(P)$ — let $\mathit{balance} := \mathsf{Win}(V, V') + \mathit{transfers}$. For $\widehat{P} \in \beta.\mathsf{end\text{-}users}$ send $\mathit{balance}(\widehat{P})$ coins to $\widehat{P}$'s account on the ledger together with a message $(\texttt{lc-closed})$, and close the contract.
   b) In time τ party P' replies with a message $(\texttt{vc-active}, z)$, where z is an opening certificate of P on some channel γ constructed over β and $\tau \leq \gamma.\mathsf{validity} + 5\Delta$ — in this case do nothing.
   c) You receive a message $(\texttt{timeout})$ from P in time $\Delta + 1$ after you sent the $(\texttt{lc-closing})$ message to P' — let $\mathit{balance} := \gamma^P.\mathsf{balance} + \mathit{transfers}$. For $\widehat{P} \in \beta.\mathsf{end\text{-}users}$ send $\mathit{balance}(\widehat{P})$ coins to $\widehat{P}$'s account on the ledger together with a message $(\texttt{lc-closed})$, and close this contract instance.

(C) Subroutine for closing a virtual channel when cheating by party P is detected:

Let $x := \gamma.\mathsf{balance}(\gamma.\mathsf{Alice}) + \gamma.\mathsf{balance}(\gamma.\mathsf{Bob})$. Remove x coins from P's account in *transfer* and add x coins to $\beta.\mathsf{other\text{-}party}(P)$'s account in *transfer*. Mark γ as closed. Send a message $(\texttt{vc-closed})$ to both $\beta.\mathsf{end\text{-}users}$.

Fig. 3: Functionality $\mathcal{C}$. For an explanation of (A) see the beginning of Sect. IV, for (B) see Sections IV-A and IV-B, and for (C) — Sect. IV-2.

References

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

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

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

[4] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd Annual Symposium on Foundations of Computer Science*. Las Vegas, Nevada, USA: IEEE Computer Society Press, 2001, pp. 136–145.

[5] Ran Canetti et al. "Universally Composable Security with Global Setup". In: *TCC 2007: 4th Theory of Cryptography Conference*. Vol. 4392. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer, Heidelberg, Germany, 2007, pp. 61–85.

[6] Christian Decker and Roger Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *SSS 2015*. Springer International Publishing, 2015, pp. 3–18. URL: http://dx.doi.org/10.1007/978-3-319-21741-3_1.

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

[8] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Vol. 1. Cambridge, UK: Cambridge University Press, 2001, pp. xix + 372.

[9] 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.

[10] Ethan Heilman et al. *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.

[11] Ahmed E. Kosba et al. "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.

[12] Joshua Lind et al. "Teechan: Payment Channels Using Trusted Execution Environments". In: *CoRR* abs/1612.07766 (2016). URL: http://arxiv.org/abs/1612.07766.

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

[14] Andrew Miller et al. "Sprites: Payment Channels that Go Faster than Lightning". In: *CoRR* abs/1702.05812 (2017). URL: http://arxiv.org/abs/1702.05812.

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

[16] 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.

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

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

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

Appendix

A. Implementation

We created a simple proof of concept implementation of the functionality $\mathcal{C}$. We wrote it in Ethereum using the programming language Solidity. The source code is publicly available on github.com/PerunEthereum/Perun.

Our main goal was to illustrate feasibility of our protocols and the underlying smart contracts. To this end, our implementation follows closely the protocol from Fig. 3. More precisely, the `LedgerChannel` contract from the implementation corresponds to the ledger channel and its functions correspond to messages of the functionality $\mathcal{C}$ from Fig. 3. The contact uses as subroutine an external contact called `LibSignatures`. The role of this contract is that it provides verification of signatures which is not available a-priori in Solidity.

Recall that when an Ethereum user sends a transaction to the Ethereum network he has to pay a transaction fee. The fees in Ethereum are paid for contract creation, execution of every operation and for storage. Fees are paid via an internal currency called *gas*, where there is an exchange rate between Ethereum's currency Ether and the internal fee currency gas. A sender of a transaction chooses the exchange rate, typically between $2 \cdot 10^{-9}$ Ether (in this case the transaction waiting time is about 5 minutes) and $2 \cdot 10^{-8}$ Ether (in this case the waiting time is about 30s). In our calculation we use the exchange rate 1 gas = $4 \cdot 10^{-9}$ Ether corresponding to a waiting time of about 50s.

In our proof-of-concept implementation the transaction fees are rather high. The highest fee is paid for `LedgerChannel` contract creation — it costs about 0.011 Ether. Fortunately, contract creation only has to be done once when the channel is set-up. The fees for calling contract functions vary between 10^{-5} and 10^{-4} Ether. We emphasize that in our prototype implementation we were not aiming at optimizing transaction.

There are several ways in which we could lower the fees. For instance, we could significantly lower fees by creating a contract LedgerChannelsSpace, that has to be deployed only once, and all LedgerChannels could be created inside it by calling its functions. We leave further optimizing transaction fees as an important question for future work.

B. Further related works

Various other micropayment schemes have been constructed in [19, 17, 13, 15, 6, 12]. 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. A very interesting construction for creating chains of ledger channels has recently been proposed in [14]. Their focus is on different aspect of channel creation than us, namely they do not attempt to remove the interaction with the intermediaries, but on making th pessimistic time of channel closing constant (which is better than the linear time in [16, 18]). It would be interesting to combine virtual channels with the techniques developed in [14]. For an overview of the techniques for dealing with the scalability problem in blockchain-based cryptocurrencies see, e.g., [12].

C. Extensions and future work

In order to keep the exposition as simple as possible, we did not include in the description of our protocol several optimizations. We now briefly describe them below (their complete formalization will appear in the extended version of this work). The first of these natural improvements is as follows. In our scheme every party P needs to remember the closing certificates of *all* the virtual channels (over some ledger channel β) that were closed in the past. This is because it may happen that long time after a virtual channel is closed a malicious $\beta.\mathsf{other\text{-}party}(P)$ party attempts to close it again, and P has to react by sending the closing certificate. This problem can be solved by adding a time limit ($\gamma.\mathsf{validity}+5\Delta+1$) to when a user can ask the channel contract to close a virtual channel. Such a restriction will not harm the honest users (as anyway every virtual channel has to be closed before that time). Hence, parties will only have to remember the closing certificates for limited amount of time.

An additional shortcoming of our construction is the fact that the intermediaries need to block the coins that are used for constructing virtual 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 notion of "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 the cheating 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 transactions of small financial value.

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) [16]. Observe also that sometimes it can make sense to allow more generous network delay times to small users, than to the payment hubs. In our simplified presentation the "waiting time" Δ is the same for all the users), but technically there will be no problems with having different waiting times for different users.

One natural question is if one can have longer virtual channels that are (a) longer, (b) have a state (see [1] for an introduction to state channels). 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 obtaining virtual channels of arbitrary length. Formally modeling this idea, constructing protocols, and proving security is beyond the scope of this work is contained in a subsequent paper.

D. The financial ledger functionality

The financial ledger functionality has already been described informally in Sect. III. Formally it is depicted on Fig. 4. It is initiated by the environment $\mathcal{Z}$ that preloads the parties with coins. The financial operations are performed by the ideal functionalities Channels and $\mathcal{C}$ by sending "add" and "remove" messages. We say that a functionality Channels or $\mathcal{C}$ *adds y coins to* (or *removes y coins from*) P's account on the ledger if it sends a message $(\underline{\texttt{add}}, P, y)$ (or, $(\underline{\texttt{remove}}, P, y)$, respectively), to the financial ledger functionality. Recall also (see Sect. III) that we assume that the simulator is allowed to freely remove money from the accounts of corrupt parties and to add them to the accounts of other (corrupt or honest) parties. This is used in the simulation, e.g., when a corrupt party got caught on cheating and all the coins in a channel get transferred to the other (honest) party (see Fig. 3 (C)).

E. Restrictions on the environment

Here we list the restrictions on the environment that were already described informally in Sect. III-C.

1) The environment $\mathcal{Z}$ never asks the parties to open a channel γ such that $\gamma.\mathsf{id}$ already exists in Σ.
2) The environment $\mathcal{Z}$ never asks the parties to open channels (ledger or virtual) when they have not enough funds available.
3) If the environment $\mathcal{Z}$ asks the parties to open a virtual channel γ then the channel with identifiers specified in $\gamma.\mathsf{subchan}$ exists in Σ, and no closing procedure for them has been initiated.
4) The environment never asks to close a ledger channel in time earlier than $\gamma.\mathsf{validity}+5\Delta$ where γ is a virtual channel whose opening has been initiated by the environment (even if this opening was unsuccessful).

Functionality $\mathcal{L}$

The global functionality $\mathcal{L}$ is running with a set $\mathcal{P}$ of parties $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$. It accepts queries of the following types:

Initialization

The functionality is initialized by a message $(x_1, \ldots, x_n) \in \mathbb{R}^n_{\geq 0}$ that describes the initial coin distribution and comes from $\mathcal{Z}$. The functionality stores this tuple.

Adding coins

Upon receiving a message $(\underline{\texttt{add}}, P_i, y)$ (for $P_i \in \mathcal{P}$ and $y \in \mathbb{R}_{\geq 0}$) let $x_i := x_i + y$.

Removing coins

Upon receiving a message $(\underline{\texttt{remove}}, P_i, y)$ (for $P_i \in \mathcal{P}$ and $y \in \mathbb{R}_{\geq 0}$): if $x_i < y$ then do nothing, and otherwise let $x_i := x_i - y$.

Fig. 4: The financial ledger functionality $\mathcal{L}$.

5) If the environment $\mathcal{Z}$ asks one of the parties $P \in \delta.\mathsf{all\text{-}users}$ to open a (ledger or virtual) channel δ (by sending a message $\underline{\texttt{lc-open}}$, or $\underline{\texttt{vc-open}}$, respectively), then it ask all the other parties in $\delta.\mathsf{all\text{-}users}$ to do the same (by sending the same message to them in the same round).
6) The environment $\mathcal{Z}$ does not perform (or confirm) any update procedures for channels whose closing has been initiated.
7) If a previous update of a channel δ failed, then $\mathcal{Z}$ will not request a new update of δ.
8) For every channel δ that the environment $\mathcal{Z}$ asks to open the identifier $\delta.\mathsf{id}$ is unique.
9) The environment always confirms an update that it initiated, and never confirms an update that she did not initiate, i.e., $\mathcal{Z}$ sends to P' a message $(\underline{\texttt{update-ok}})$ as a reply to $(\underline{\texttt{update-requested}}, id, \theta, \alpha)$ if only only if 1 round earlier $\mathcal{Z}$ did not send $(\underline{\texttt{update}}, id, \theta, \alpha)$ to P (cf. Fig. 5, Point (D)).

A consequence of these restrictions is that in our protocol we can assume that all the honest parties have the same view on what channels should be open. For example: $\beta.\mathsf{Alice}$ knows that if she received a $(\underline{\texttt{vc-open}}, \beta)$ message from $\mathcal{Z}$ then $\beta.\mathsf{Bob}$ also received such a message (in the same round). This, in particular, means that if $\beta.\mathsf{Bob}$ refuses to participate in the procedure of opening channel β then she must be corrupt.

F. Formal security definition

To formally define security we define two random variables denoting the output of $\mathcal{Z}$ in the real and ideal world. More precisely, we denote the output of the environment $\mathcal{Z}$ in the real world execution of a protocol `channels` in presence of an ideal functionality $\mathcal{C}$ and an adversary $\mathcal{A}$ by $\text{EXEC}^{\mathcal{Z},\mathcal{A}}_{\texttt{channels},\mathcal{C}}(\lambda)$. Here λ denotes the security parameter. Notice that $\text{EXEC}^{\mathcal{Z},\mathcal{A}}_{\texttt{channels},\mathcal{C}}(\lambda)$ is in UC terminology often also called the $\mathcal{C}$ hybrid world. In the ideal world, the parties simply forward their inputs to the ideal functionality $\mathsf{Channels}$ (see Fig. 5). The output of the environment after an ideal execution against an ideal-world adversary $\mathcal{S}$ (often called a *simulator*) with security parameter 1^λ is in this case denoted $\text{IDEAL}^{\mathcal{Z},\mathcal{S}}_{\mathsf{Channels},\mathcal{S}}(\lambda)$. We are now ready to state our main security definition.

Definition 1: We say that a protocol `channels` and an ideal functionality $\mathcal{C}$ implements the ideal functionality $\mathsf{Channels}$ if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every environment $\mathcal{Z}$ (from the class described in Sect. III-C) we have that

$$\{\text{EXEC}^{\mathcal{Z},\mathcal{A}}_{\texttt{channels},\mathcal{C}}(\lambda)\}_\lambda$$

is computationally indistinguishable from

$$\{\text{IDEAL}^{\mathcal{Z},\mathcal{S}}_{\mathsf{Channels},\mathcal{S}}(\lambda)\}_\lambda.$$

G. Security analysis

In the section we prove sketch the proof of the following theorem (its complete proof will be provided in the extended version of this paper).

Theorem 1: The protocol `channels` and an ideal functionality $\mathcal{C}$ constructed in Sect. IV implement the ideal functionality $\mathsf{Channels}$.

Proof sketch. We have already informally argued about the security of our scheme while presenting it in Sect. IV. Here we focus on describing the simulator $\mathcal{S}$ for some fixed adversary $\mathcal{A}$. Recall that $\mathcal{S}$ interacts with the environment $\mathcal{Z}$ and the ideal functionality $\mathsf{Channels}$ (via the so-called "dummy" parties, see [4]), and its goal is to "emulate" the behavior of $\mathcal{A}$ for the environment.

At the beginning the simulator $\mathcal{S}$ starts the adversary $\mathcal{A}$ and corrupts the parties that $\mathcal{A}$ corrupts (for simplicity we assume that $\mathcal{A}$ is static, i.e., he decides whom to corrupt at the beginning of the execution of the protocol). The simulator also generates the (public key, private key) pairs for all the users. He passes the public keys of all the users to the corrupt users, and to each corrupt P_i he also sends his private key sk_i. Then $\mathcal{S}$ simulates the behavior of $\mathcal{A}$, and watches the instructions of $\mathcal{A}$ to the corrupt parties. Depending on the behavior of the simulated $\mathcal{A}$ the simulator sends inputs of his choice to the $\mathsf{Channels}$ functionality.

To make it impossible to distinguish between the simulated and the real execution, the simulator needs to emulate the messages sent to the corrupt parties by the $\mathcal{C}$ functionality and by the other (honest) parties. Recall also that the adversary $\mathcal{A}$ "controls the network" meaning that he decides when the

(A) **Opening a ledger channel**:

Upon receiving a message $(\underline{\texttt{lc-open}}, \beta)$ from β.Alice with β.balance(β.Alice) coins in round τ, where β is a ledger channel, proceed as follows:

1) Within round $\tau + \Delta$ remove $x_A := \beta$.balance(β.Alice) coins from β.Alice's account on the ledger $\mathcal{L}$.
2) If within time Δ after Step 1 was completed you receive a message $(\underline{\texttt{lc-open}}, \beta)$ from party β.Bob, then remove $x_B := \beta$.balance(β.Bob) coins from β.Bob's account on the ledger $\mathcal{L}$, and add β to Σ. Output ($\underline{\texttt{lc-opened}}$) to parties in β.end-users and to the simulator $\mathcal{S}$, and stop.
   Otherwise within time 2Δ after Step 1 was completed add x_A coins to the account of β.Alice on the ledger $\mathcal{L}$ and ($\underline{\texttt{lc-not-opened}}$) to β.Alice.

(B) **Opening and closing a virtual channel**:

1) Upon receiving a message $m = (\underline{\texttt{vc-open}}, \gamma)$ (where γ is a virtual channel) from *all* the parties in γ.all-users (within 2 rounds), do the following:
   a) for $P \in \gamma$.end-users remove γ.balance(P) coins to P's account in $\Sigma(\gamma.\text{subchan}(P))$.
   b) remove γ.balance(γ.Bob) coins from the account of γ.Ingrid in $\Sigma(\gamma.\text{subchan}(\gamma.\text{Alice}))$, and γ.balance(γ.Alice) coins from the account of γ.Ingrid in $\Sigma(\gamma.\text{subchan}(\gamma.\text{Bob}))$.

   Then add γ to Σ, output ($\underline{\texttt{vc-opened}}$) to the parties in γ.all-users, and go to Step 2.
   If within 2 rounds (from receiving m for the first time) you do not receive m from *all* the parties in γ.all-users then output $\underline{\texttt{vc-not-opened}}$ to them and stop.
2) Wait until round γ.validity (in the meanwhile accepting the "channel update" requests that concern γ, see below). When this round comes let $\widehat{\gamma} := \Sigma(\gamma.\text{id})$ be the current version of γ, and execute the following operations within round γ.validity $+ 5\Delta + 1$:
   a) for $P \in \gamma$.end-users add $\widehat{\gamma}$.balance(P) coins to P's account in $\Sigma(\gamma.\text{subchan}(P))$.
   b) add $\widehat{\gamma}$.balance(γ.Bob) coins to the account of γ.Ingrid in $\Sigma(\gamma.\text{subchan}(\gamma.\text{Alice}))$, and $\widehat{\gamma}$.balance(γ.Alice) coins to the account of γ.Ingrid in $\Sigma(\gamma.\text{subchan}(\gamma.\text{Bob}))$,

   Output ($\underline{\texttt{vc-closed}}$) to the parties γ.all-users and erase $\widehat{\gamma}$ from Σ.

(C) **Ledger/virtual channel update**:

Upon receiving a message $m := (\underline{\texttt{update}}, id, \theta, \alpha)$ (such that there exists a channel $\delta \in \Sigma$ with identifier id) from a party $P \in \delta$.end-users. Send a message $(\underline{\texttt{update-requested}}, id, \theta, \alpha)$ to $P' := \delta.\text{other-party}(P)$.
Once P' replies with a message ($\underline{\texttt{update-ok}}$) replace δ in Σ with a channel $\widehat{\delta}$ that is equal to δ, except that $\widehat{\delta}$.balance $:= \delta$.balance $+ \theta$ and send ($\underline{\texttt{updated}}$) to P.

(D) **Ledger channel closing**:

Upon receiving a message $(\underline{\texttt{lc-close}}, id)$ (such that there exists a ledger channel $\beta \in \Sigma$ with identifier id) from a party $P \in \beta$.end-users do the following:

1) If there exists open virtual channels built over β, then ignore this message.
2) Otherwise within time Δ add β.balance(β.Alice) coins to β.Alice's account on $\mathcal{L}$, and β.balance(β.Bob) coins to β.Bob's account on $\mathcal{L}$, respectively. Erase β from Σ and send ($\underline{\texttt{lc-closed}}$) to the parties in β.end-users and to the adversary.

Fig. 5: Functionality Channels run by a set of parties $\mathcal{P} := \{P_1, \ldots, P_n\}$ and an adversary $\mathcal{A}$. The functionality maintains a channel space Σ that is initially empty. See Sect. III-B for a informal description of these procedures.

messages are delivered, subject to some timing constraints. In particular, we assumed that sending message to $\mathcal{C}$ takes time at most Δ. In our protocol the honest parties always send messages to $\mathcal{C}$ early enough so that they reach $\mathcal{C}$ before its "too late" (e.g. an honest β.Bob always sends the lc-opening message to $\mathcal{C}$ immediately after receiving message lc-open from β.Alice in Step 2.2 on Fig. 1 (A)). On the other hand, the corrupt parties, may send such messages at any time they want. Therefore our simulator has to observe the network and watch how much delay $\mathcal{A}$ introduces when delivering a given

message m to $\mathcal{C}$, and based on this decide whether m was delivered "on time" or not.

Below we describe how different parts of the channels protocol are handled by the simulator. Observe that the only non-trivial cases are when some of the parties participating in a given part of the protocol (eg.: β.end-users or γ.all-users) are corrupt and some are honest. This is because the case when all of them are corrupt is easy to handle, since the simulator can just internally simulate the execution of corrupt parties, and then take care of distributing coins that result from this (recall that he has power to freely transfer coins between the honest parties). It is also easy (by inspecting the protocol) that if all the parties are honest then they perfectly emulate the execution of the ideal functionality.

a) Ledger channel opening.

This part starts when $\mathcal{Z}$ sends an $(\underline{\texttt{lc-open}}, \beta)$ message to both β.end-users in some round τ. Simulating it is straightforward: the simulator simply simulates the contract functionality, plays the role of the honest party to the corrupt one, and removes the coins from the ledger when the parties send messages with coins to the contract (and refunds this money to β.Alice if the channel is not open).

b) Channel updating.

The part for updating a ledger or virtual channel δ starts when $\mathcal{Z}$ sends an $m = (\underline{\texttt{update}}, id, \theta, \alpha)$ message to the update initiator $P \in \delta$.end-users. If P is corrupt then $\mathcal{S}$ sends m to P. If in the next round P sends the updating message to $P' := \delta$.other-party(P) (with all the parameters computed correctly) then $\mathcal{S}$ sends m (in the name of P) to the ideal functionality Channels. Then in the next round $\mathcal{S}$ sends to P the confirmation message from P' (recall that by Restriction 9 we assumed that the environment always confirms such updates). Note that this requires signing messages with P' private key, but $\mathcal{S}$ can do it, since he knows the private keys of all the parties.

Simulating the update procedure is a bit more tricky in case when the initiator P is honest and the confirmer P' is corrupt. This is because the confirmer may not send his signature on the updated channel state back to P, but, since he already knows P's signature on it, he can use it when the channel is closed.

We distinguish two cases. The first case is when the transfer is beneficial to P', i.e., $\theta(P') > 0$. In this case we will assume that even if P' does not immediately send her signature on the updated state to P then the update did happen. Intuitively, this is because any "rational" P' will use this new updated state during the channel closing. Hence the simulator sends $(\underline{\texttt{update-ok}})$ in the name of P' to the ideal functionality Channels, no matter if P' sends his signature on the new state to P or not. Of course, an "irrational" P' can still use the old channel state when the channel is closed, and get less coins than he could get by posting the newest version of the state. This is not a problem, since the simulator can always remove these extra coins from the account of P' and move them to the account of P immediately after channel closing has been performed.

The case when $\theta(P') \leq 0$ is symmetric. In this case, if the simulated corrupt P' does not send his confirmation on the update immediately to P, then the simulator does not send (in the name of P') the message $(\underline{\texttt{update-ok}})$ to the ideal functionality Channels. In other words, we make P conclude that P' did not accept a transfer that was beneficial to P. Again, it can happen during the channel closing that P' will use the newer version ("against his own financial interest"). This again can be corrected by the simulator $\mathcal{S}$ transferring the coins from P' to P immediately after channel closing.

c) Ledger channel closing.

This part starts when $\mathcal{Z}$ sends to P a message $(\underline{\texttt{lc-close}}, id)$. It can also be started by a corrupt user of a ledger channel β whenever the adversary instructs him to do so. As in the case of channel opening the simulation is also straightforward: the simulator simply simulates the other parties and the contract functionality for the corrupt party, and sends the lc-close message Channels once P successfully closes a channel. The only thing that we need to remember is that if a corrupt $\widehat{P} \in \beta$.end-users may submit a version of a channel that is less beneficial for him, but newer, than the latest version that the other user of β submits (see above). As described above, in this case $\mathcal{S}$ simply moves the appropriate amount of coins from $\widehat{P}$'s account in the ledger to the account of β.other-party$(\widehat{P})$ to "correct" this difference.

d) Virtual channel opening

This part starts when $\mathcal{Z}$ sends a message $(\underline{\texttt{vc-open}}, \gamma)$ to all parties in γ.all-users. The simulation proceeds as in the previous cases, i.e., the simulator simulates the behavior of the corrupt parties, and emulates the honest parties for them. When the simulated honest parties output a $(\underline{\texttt{vc-opened}})$ message then the simulator sends the $(\underline{\texttt{vc-open}}, \gamma)$ message in the name of corrupted $P \in \gamma$.all-users to the ideal functionality Channels and lets the ideal functionality immediately output $(\underline{\texttt{vc-opened}})$ to all the users. This is ok, since, as we argued while presenting the protocol, the honest parties will always agree on whether the channel has been opened or not.

e) Virtual channel closing

Closing of virtual channel starts automatically when time γ.validity comes. As argued in Sect. IV-B the virtual channel is always closed, as long as at least one party on γ.all-users is honest. Again, the simulator simulates the corrupt parties, and, depending on their behavior instructs the ideal functionality Channels to send the $(\underline{\texttt{vc-closed}})$ message in the right moment to the honest parties (in time at most γ.validity$+5\Delta+1$).

In the extended version of this paper we will provide a complete description of the simulator, and the argument why the output $\{\text{IDEAL}^{\mathcal{Z},\mathcal{S}}_{\mathsf{Channels},\mathcal{S}}(\lambda)\}_\lambda$ in the ideal world is computationally indistinguishable from the real-world output $\{\text{EXEC}^{\mathcal{Z},\mathcal{A}}_{\texttt{channels},\mathcal{C}}(\lambda)\}_\lambda$. □