cf-specs Documentation

Counterfactual Contributors

Dec 12, 2019

Contents:

CHAPTER 1

Introduction

Counterfactual is a framework for building off-chain state-channel-based applications. State channels allow users to interact with each other without paying blockchain transaction fees and with instant finality. Channelization is the only technique that securely realises the latter property.

In the protocol, participants exchange cryptographically signed messages which are pre-signed transactions that distribute the blockchain state or perform other tasks necessary for the channel's correct outcome. Next, participants deposit blockchain state into an n-of-n multisignature wallet referenced by the transactions. New cryptographically signed state updates related to the original commitments can now be used to change the state and / or assets controlled by the multisignature wallet.

Through a challenge-response mechanism, on-chain contracts implement methods for participants to ensure that the latest signed valid state update that pertains to their commitment can be submitted to the blockchain, guaranteeing correct outcome of the state for all users adhering to the protocol.

1.1 Framework Design Goals

The Counterfactual framework is still a work in progress. Its current design (and future roadmap) are driven primarily by the following criteria:

1.1.1 Minimized on-chain footprint

We don't want to put anything on the chain that doesn't need to be. We aim to make a generic multisignature wallet the only necessary on-chain object that needs to be deployed for a new state channel.

1.1.2 Ease-of-use

We want channels that can be easily incorporated into new applications without the requirement that their developers also be state channel experts. To provide at least one such simple method for developers to utilize within our framework, we provide an abstraction for state-machine-based channel applications, or "Apps". This class of "App" consists of simple stateless contracts which define a state machine, including valid transitions and turn-taking logic. Although

state-machine-based "Apps" are an intentionally restricted subset of state channel functionality, they nonetheless enable developers to deploy a wide range of channelized applications without butting up against the often complex and subtle *limitations* of state channel design. As the protocol develops further more complex functionality will continue be added, allowing easy utilization of increasingly advanced techniques by developers using the Counterfactual framework.

1.1.3 Standardized

We want to establish clear standards for how all of these generalized state channels will fit together into a global, multi-blockchain network where any user can easily connect to any other. To achieve this goal, we work closely with great researchers from Celer, Magmo, Ethereum Research and several others. We hope to amplify these efforts to work towards blockchain standards for off-chain channelized applications more broadly.

1.1.4 Parallel operations

We want to support multiple parallel operations inside of a single channel that do not interfere with each other. We have designed state-machine-based "Apps" to maintain control of the state assigned to them in a fashion completely independent of each other. Typical operations like installing new applications, uninstalling old applications, and updating applications are fully parallelized operations within the protocol.

1.1.5 Upgradeable

We want to support deploying or upgrading channel designs without requiring the user to make a single on-chain operation. There are multiple techniques which are specifically anticipated in the current design. For the purposes of *trustless* off-chain upgradability, we are able to support counterfactually instantiated smart contracts as applications. To upgrade a contract trustlessly, state channel participants can simply agree off-chain to a new version of bytecode for their application. At the cost of certain additional trust assumptions, state channel participants could also use an application that is defined using ZeppelinOS's upgradeable contracts or a similar method.

1.1.6 Maximized privacy

We want to achieve a level of privacy where state channel operations are indistinguishable from other common types of on-chain activities. Using a state channel should not reveal any information about the applications that are being used, the state being used within them, or even the fact that a state channel is being used at all. As a first step towards preserving this property, we assume that the on-chain component is a generic multisignature wallet which looks the same as any other multisignature wallet on Ethereum. In the future we expect that stricter levels of privacy will be enabled by various zero knowledge constructions, and that those will fit best when applied in similarly general, abstract ways that fit neatly with this approach.

1.2 Protocol Design Goals

1.2.1 Constant sized communication

The number of messages an message sizes for an operation are independent of

- Total number of active off-chain applications
- Total number of inactive off-chain applications
- Total number of sequential state updates to an application

That is to say, the design aims for **parallelizability** in general, ensuring that historical use of the protocol does not impact the size of messages being transmitted on future use of the protocol.

1.2.2 O(1) response to stale state

It is possible to arrive at a state where any placement of stale state on chain can be responded to with a single transaction of constant size, in particular, independent of number of active or historical apps. This goal is to ensure that any kind of inevitable griefing vectors that are impossible to fully disqualify off-chain are resolvable with the minimum amount of cost to the person being griefed on-chain.

CHAPTER 2

State Channel Applications

When we discuss building off-chain applications in general, we usually reference easy-to-understand examples such as payment channels, turn-based games like Tic-Tac-Toe, or well understood blockchain use cases like mixers. For each of these examples, we attempt to isolate their core functionality inside of a single logical container. We call these containers **Apps**. Apps have the following properties at a minimum:

- The number of participants / users supported
- An encoding type for its state
- An function that returns an outcome from a state

The most basic type of App is a 2-person ETH payment channel. In that case we have:

- Alice and Bob as the users
- An encoding type of `tuple(address aliceAddr, address bobAddr, uint256 aBal, uint256 bBal)`†
- The outcome function sends `aBal` Wei to `aliceAddr` and sends `bBal` Wei to `bobAddr`

A slightly more complicated example would be a Tic-Tac-Toe game:

- Alice and Bob are the users, Alice is X and Bob is O
- An encoding type of `tuple(uint8[9] board, address playerX, address playerO)`
- Sends `playerX` the maximimum amount if X won, or `playerO` if O won, or splits the amount in half if a draw

There is, of course, an important difference in the example of a Tic-Tac-Toe game though. That difference is that in the Tic-Tac-Toe game, the final outcome of the state of the application is not always defined given only the current state of the application. In the payment channel example, if one user were to become unresponsive it is easy to see how the outcome would play out; Alice would receive `aBal` and Bob would receive `bBal`. In Tic-Tac-Toe, however, if the game is not finished yet, there are some moves left to be made to reach a "terminal" state which can then be used to reach an outcome. Terminal states comprise those in which X wins, O wins, or there is a draw.

To express this important difference we introduce some additional functionality that can be implemented for a state channels based application:

- A function which defines how state can "progress" given an action
- A function to define whether an action can be legally taken by a participant
- A function to determine is a state is indeed one that could be considered "terminal"

For Tic-Tac-Toe then these can be expressed as:

- Allowed actions are to place an X on the board or place an O on the board
- `playerX` may place an X if it is X's turn based on the `board` state (and vice versa)
- The state is "terminal" if there are 3 in a row of X's or O's on the board or the board is full

In the next sections, we define how in Counterfactual we define application state, progress it, and use it to arrive at outcomes.

2.1 Defining State

A state channel is fundamentally about progressing a single state object. Therefore, in the Counterfactual framework, we center everything around a single `bytes` object. For the sake of performing computation on this object (e.g., evaluating if there is a row of X's on a Tic-Tac-Toe board) we interpret it as an ABI-encoded value of a specific type using the built-in `ABIEncoderV2` language feature. THIS allows developers to define their state structures using a `struct` definition and encode and decode these objects as needed. For example, in our payment channel from above we might have the following object:

```
struct ETHPaymentChannelState {
  address alice;
  address bob;
  uint256 aBal;
  uint256 bBal;
}
```

2.2 Progressing State

As has been mentioned before, some kinds of applications require a way of progressing some state to a "terminal" state through a series of allowed actions. In these cases, we adopt the model of a state machine that consists of logical states and allowed actions (transitions between states); a "terminal" state is simply one from which there does not exist any outgoing edge (i.e., an "allowed action").

In Counterfactual, if an action wishes to allow its state to be unilaterally progressable, we require the definition of a function that **applies an action to a state to produce a new state** *in addition to* **a function that determines if an action can be taken by a particular turn taker**. As you will see in the *adjudication layer* section of these specifications, these functions are important in handling on-chain challenge scenarios.

The ultimate purpose of these functions is to ensure the following:

- It should always be extremely explicit what the exact rules of the state channel application that all parties are abiding by are
- There should always be a single logical turn taker for any given state (*concurrent state updates* are disallowed)
- It should be possible, in the least number of on-chain transactions, use the adjudication layer to fairly arrive at an outcome for a state channel application (without unnecessary gas expenditure)

Here is a helpful diagram for visualizing the nature of such an `applyAction` function:

2.3 Resolving State

A state channel application defines a outcome. This is a critical concept usually because the outcome can be tied to interesting economic parameters that create the incentives for the behaviour of users in the application to begin with. For example, users will remain online and play a game of Tic-Tac-Toe because they know the rules of the game are deciding who will take home some financial reward. In the Counterfactual framework, this reward is defined in terms of an internal transaction that is executed by the multisignature wallet that is the holder of the state deposits.

2.4 AppDefinition

To address all of the above requirements of state channel applications, we introduce an interface called an `AppDefinition` which **implements the logic of an application in the EVM**. The `AppDefinition` interface is implemented by a developer interested in writing a state channels based application that the Counterfactual project supports in the rest of the framework (e.g., in the *adjudication layer*).

2.4.1 Outcome Function

There is a single function which *must* be implemented in the interface. This function provides the outcome functionality discussed above. The function signature is:

```
function computeOutcome(bytes memory) public view returns (bytes);
```

The first argument of type `bytes memory` is an internally-referencable state object for the `AppDefinition`. For example, in the case of a payment channel it must be considered as the ABI-encoded version of a structure encoding the type from above. This means that you will want to use `abi.decode` inside of the `computeOutcome` method to decode the bytes into a usable struct object. In our payment channel example this would look like:

```
function computeOutcome(bytes memory encodedState)
  public
  view
  returns (bytes)
{
  AppState state = abi.decode(encodedState, (ETHPaymentChannelState));
  // state.aBal, state.bBal, state.alice, state.bob
}
```

2.4.2 Outcome Type

What should be returned from the `computeOutcome` function depends on the **outcome type** of the app definition. A outcome type defines the possible outcomes of the app instance and what effects are compatible with it. For instance, a two-player chess game has "categorical" outcomes of win/loss/draw, whereas a two-player poker game with 2 ETH total pot has "continuous" outcomes where the first player may be allocated any amount from 0 to 2 ETH. The different nature of the applications mean that a state deposit of ERC721 non-fungible tokens can be used with chess but not with poker.

In the framework, the following outcome types are currently defined.

- TwoPlayerOutcome: the app results in an outcome of a win, a loss, or a draw. This is represented as the abi encoding of a `uint256`.
- CoinTransfer: the app arrives in an outcome of a dynamically-sized mapping of address to amount in Wei. This is represenetd as the abi encoding of `tuple(address, uint)[]`.

2.4.3 Optional Functionality

In addition to the mandatory `computeOutcome` method, the `AppDefinition` interface optionally allows for the definition of the following methods.

2.4.4 Turn Taking Function

To identify who is uniquely allowed to progress from one state to the next, a turn taking function can be implemented which returns the specific address that is expected to have signed a particular state update. It accepts two arguments: the state of the application and the array of all participants that have been allocated to the application. Therefore, the function must return the key in the array of participants that it expects to sign an update.

```
function getTurnTaker(bytes memory, address[] memory) public pure returns (address);
```

In our Tic-Tac-Toe example, this function would return the 0th-indexed key for player X and the key at index 1 for player O.

2.4.5 Terminal State Flag Function

As an efficiency gain in cases where the adjudication layer is needed, a function can be defined which declares whether or not a state can no longer be progressed. This function takes a single argument: the state of the application. It is expected to return `true` if the state is terminal or `false` if it is not.

```
function isStateTerminal(bytes memory) public pure returns (bool);
```

In Tic-Tac-Toe, the state is terminal if the game has been won or the board is full.

2.4.6 Action Application Function

Finally, the most critical function for progressing state is the `applyAction` function which as described above takes some encoded state and an encoded action and returns a new encoded state object. The *encoding and decoding* functionality provided in Solidity are helpful here.

```
function applyAction(bytes memory, bytes memory) public pure returns (bytes memory);
```

In Tic-Tac-Toe, this function would place the X or the O on the board based on the action type if the position on the board is not already filled.

2.5 Footnotes

2.5.1 ABIEncoderV2

We use `ABIEncoderV2` in the framework to represent arbitrary EVM-compatible state

There are helpful functions that are offered in Solidity which can be used for encoding and decoding.

Encoding:

```
ExampleStruct state = ExampleStruct(...);
bytes encodedState = abi.encode(state);
```

Decoding:

```
bytes encodedState; // 0x.....
ExampleStruct state = abi.decode(encodedState, (ExampleStruct));
```

2.5.2 Concurrent State Updates

Presently in Conterfactual progressable state machines must be uniquely progressable by a single turn taker and must move one turn at a time. The reason for this requirement is that if two or more users were able to progress the state of an application independently, it is possible that they may enter into a conflicting state.

As an example of this, consider an application where two users can move the the (x, y) co-ordinates of two pieces on a 10x10 grid; each can move one but not the other. In this example, it is possible that if they were able to move their pieces concurrently they could enter into a state where two pieces are in the same (x, y) co-ordinate which may be disallowed in the logic of the application. In that case a race condition occurs where whoever is able to submit their transaction to the blockchain first would be able to consider their state as final and the other would be disallowed. Of course, this is a kind of behaviour we want to avoid when designing state channel applications.

There are scenarios, however, where you *want* concurrent updates and it is indeed possible to allow them. The general class of data structures that represent these types of data structures are called conflict-free replicated data types (CRDTs) and are commonly used in distributed systems.

In a future version of the `AppDefinition` interface, we hope to support a version of these kinds of objects such that concurrent state updates might be possible.

CHAPTER 3

Adjudication Layer

Counterfactual's adjudication layer uses a singleton contract called the `ChallengeRegistry`. This contract has been designed to only be compatible with applications that implement the `CounterfactualApp` interface.

The adjudication layer treats off-chain state as divided into independent app instances, which are "instantiations" of Apps governed by the app definition. A channel with two chess different chess games ongoing has two different app instances.

Three core concepts of the adjudication layer are **challenges**, **outcomes** and **interpreters**. Challenges are the adjudication layer's mechanism to *learn the final state* of an off-chain application. This final state is stored as an outcome. Interpreters use the outcome in order to *distribute state deposits*. As a concrete example, a challenge might be about the state of the board in a game of Tic-Tac-Toe, the outcome about who has won the game, and the interpreter produce the effect of sending 2 pre-committed ETH to the winner.

3.1 AppInstance and AppIdentity

An app instance is uniquely identified by its `AppIdentity`.

```
struct AppIdentity {
  uint256 channelNonce;
  address[] participants;
}
```

- **`participants`**: These are the keys expected to have signed any state updates that are meant to be validated by the `ChallengeRegistry`.
- **`channelNonce`**: A number used to identify uniqueness of the AppInstance

The hash of the app identity, defined as `keccak256(abi.encode(appIdentity.channelNonce, appIdentity.participants))`, uniquely identifies the app instance as well.

3.2 Challenges

A challenge that is put on-chain must result from a failure of some state channel users to follow the protocol. In most cases, it represents a failure of responsiveness.

3.2.1 Data Structure

The `ChallengeRegistry` contains a storage mapping from appIdentityHash to a struct that represents the current challenge the app instance is undergoing

```
mapping (bytes32 => LibStateChannelApp.AppChallenge) public appChallenges;
```

A challenge is represented by the following data structure:

```
struct AppChallenge {
    ChallengeStatus status;
    address latestSubmitter;
    bytes32 appStateHash;
    uint256 challengeCounter;
    uint256 finalizesAt;
    uint256 versionNumber;
}
```

Where `ChallengeStatus` is one of `NO_CHALLENGE`, `EXPLICITLY_FINALIZED`, or `FINALIZES_AFTER_DEADLINE`.

Here is a description of why each field exists in this data structure:

- **`status` and `finalizesAt` **: A challenge exists is either "not finalized" or "finalized". A timeout is used as one transition between these two states. These two fields together determine if the channel is currently finalized and if there is a timeout running.
  - `NO_CHALLENGE`: Has never been opened (the "null" challenge and default for all off-chain apps)
  - `FINALIZES_AFTER_DEADLINE`, `finalizesAt` in future: open and can be responded to
  - `FINALIZES_AFTER_DEADLINE`, `finalizesAt` in past: timeout expired, the challenge was finalized
  - `EXPLICITLY_FINALIZED`: Was finalized via a transition to an `is_final` state

statechannel statuses

- **`latestSubmitter`**: This is a record of *who* submitted the last challenge. This is useful to ensure that if a *provably malicious* challenge was submitted, the malicious party can be punished.
- **`appStateHash`**: This is the hash of the latest state of the application. We only need to store the hash of the latest state in the event of a challenge because it is possible to accept the full state as calldata, hash it in the EVM, and then compare this to the hash and keep the full state available in calldata to be used for computation.
- **`versionNumber`**: This is the versionNumber (i.e., the monotonically increasing version number of the state) at which the `appStateHash` is versioned for a particular challenge.
- **`challengeCounter`**: The `challengeCounter` is a permanent marker of how many times a challenge has been issued and re-issued on-chain for a particular AppInstance. Parties can pre-agree that if this counter ever reaches an excessively high number, simply conclude the application at some pre-determined outcome. This is simply a special mechanism to conclude applications in cases of excessive griefing by one counterparty. For more information, I recommend reading the economic risks section of the Counterfactual paper.

Since the contract that Counterfactual relies on for managing challenges is a singleton and is responsible for challnges that can occur in multiple different state channels simultaneously, it implements a mapping from what is called an `AppIdentity` to the challenge data structure described above.

3.2.2 Initiating a Challenge

Counterparty is unresponsive. In the event that one user becomes unresponsive in the state channel application, it is always possible to simply submit the latest state of the application to the `ChallengeRegistry` contract. This requires submitting an `AppIdentity` object, the hash of the latest state, its corresponding versionNumber, a timeout parameter, and the signatures required for those three data. This initiates a challenge and places an `AppChallenge` object in the storage of the contract assuming all of the information provided is adequate.

Counterparty is unresponsive *and* a valid action exists. In the case that an application adheres to the `AppDefinition` interface and provides valid `applyAction` and `getTurnTaker` functions, an action can additionally be taken when initiator a challenge. A function call to the `ChallengeRegistry` with the latest state parameters exactly as in the case above but with an additional parameter for an encoded action and the requisite signatures by the valid turn taker can be made. In this case, a challenge is added the same as above but the state is progressed one step forward.

3.2.3 Responding to a Challenge

Cancelling the challenge. In the simplest case, you and your counterparty can always sign a piece of data that declares "we would both like to cancel this challenge and resume off-chain". In this case, simply provide the contract with this evidence and the challenge can be deleted and ignored.

Progressing the challenge on-chain. If the counterparty that initiated the challenge is not willing to continue the application off-chain, but their challenge is indeed a valid one (perhaps you were offline temporarily), then you can respond to the challenge on-chain (and in doing so, issue a new challenge) if there exists an implementation for `applyAction` and `getTurnTaker`. In this case, much like is the case when initiator a challenge, you simply provide an encoded action and requisite signatures to re-issue the challenge but in the opposite direction.

Handling a malicious challenge. If the counterparty submitted stale state that is *provably malicious*, then the contract supports claiming this by submitting a provably newer state that convicts them and rewards the honest party (i.e., you in this case).

3.3 Outcomes

After a challenge has been finalized, the `ChallengeRegistry` can now be used to arrive at an outcome. In the Counterfactual protocols, it is the *outcome* of an application that is important in executing the disribution of blockchain state fairly for any given off-chain application.

3.3.1 Setting an Outcome

After a challenge is finalized. If a challenge has been finalized by the timeout expiring, then a function call can be made to the `ChallengeRegistry` that then initiates a call to the `computeOutcome` function of the corresponding `AppDefinition` to the challenge. The `computeOutcome` method will return a `bytes` struct and that is then stored inside the contract permanently as the outcome of the application.

In the same transaction as finalizing a challenge. A minor efficiency can be added here, but has not yet been implemented, which is that if the challenge can finalized unilaterally (either in initiation or in refutation) then it is possible to instantly set the outcome. There is an issue tracking this on GitHub.

3.4 Interpreters

Interpreters read outcomes in order to produce an effect. The interpreter must be compatible with the outcome type of the application as defined in the *app definition*.

Two interpreters are currently defined, although more can be added independently by add

- TwoPartyFixedOutcomeInterpreter splits a fixed amount of ETH based on a TwoPartyFixedOutcome. The parameters consist of two beneficiary addresses and the total amount of Eth, and the params are encoded as `tuple(address[2], uint256)`.
- MultiAssetMultiPartyCoinTransferInterpreter sends ETH based on an CoinTransfer outcome, up to a fixed upper bound. The paramse are encoded as `uint256`.

The interpreter used and the params to the interpreter are fixed per app instance by being included in the appIdentityHash computation. After an outcome is stored, the adjudication layer allows a commitment to `ConditionalTransactionDelegateTarget.sol:executeEffectOfInterpretedAppOutcome` call the intepreter on the outcome.

3.5 FAQ

Why are the contracts only compatible with the state channel framework?

The core concepts are agnostic to the underlying interface that a state channels application might implement. A future version of the `ChallengeRegistry` might support other types of state channel architectures such as Force Move Games, for example.

Can an outcome be part of the application state itself?

From a conceptual point of view, we think it is important to separate these two concepts. However, from an engineering point of view it can be more efficient to group the two together in a single state object. To be specific, if the outcome of an off-chain application is a agnostic to blockchain state (i.e., `pure`) operation on the state of the application, then it is safe to group the two together. However, if the outcome has a dependancy on an external contract or the block number (i.e., a `view` function) then the outcome *must* be separately computed at a later time on-chain.

Can participants sign new state off-chain during an on-chain challenge?

It is possible that an application may have a challenge initiated on-chain and then have some state updated correctly off-chain. This would likely only occur in the case of a software error, but nonetheless it is possible. In this situation, whatever state has been put on-chain at some versionNumber `k` is considered to be *more valid* than any off-chain signed state at versionNumber `k`. In order to progress the state off-chain, you would need to sign a new state with versionNumber `k + 1`.

Can we punish stale state attacks?

A challenge for an `n`-party off-chain application is considered to be provably malicious if the `versionNumber` of the challenge was `k` and someone was able to respond with a state signed by the `latestSubmitter` where the `versionNumber` of *that challenge response* was at least `k + 2`.

The reason why the same version of the above scenario with `versionNumber` equal to `k + 1` is not considered malicious is that the following situation might occur by an honest party:

- Honest party A signs state with versionNumber `k` and send it to B. Then, B countersigns and sends to A
- Honest party A signs state with versionNumber `k + 1` and send it to B, Then, B is unresponsive

In this situation, honest party A holds a signed copy of state with versionNumber `k` signed by B and can initiate a challenge on the blockchain. However, it is possible that B did in fact receive the signed state with versionNumber `k`

`+ 1` and can then respond to the challenge with this. Therefore, we must require that a state put on chain have at least `k + 2` to be considered an attempt at a stale state attack.

Where is the mapping from app definition to outcome type stored?

Currently, this is stored on the app definition contract. In the future, it should be stored off-chain.

CHAPTER 4

Channel Networks

The framework currently contains basic support for channel networks via "virtual apps". In this design, an app instance between that set of parties can have some state deposit assigned indirectly to it, without the set of parties having a direct channel containing them. A commitment is made to `ConditionalTransactionDelegateTarget.sol:executeEffectOfInterpretedAppOutcome` with the `appIdentityHash` being that of the virtual app but the interpreter is unique for each side of the network. The outcome is interpreted to send one side the value the virtual app's outcome dictates they should receive and the intermediary the other side's value. For example, in an A-I-B virtual app, where I is intermediating an app instance between A and B, then there is a commitment made in the A-I channel that interprets the outcome of the A-B virtual app such that A would receive A's result and I receives B's result.

4.1 FAQ

Why is the agreement thing specialized to `TwoPartyFixedOutcome` and `ETH`?

This is similar to why we have different interpreters - the app definition writer does not explicitly choose which assets their app supports, but simply implements an outcome type.

Are all asset types supported by this design?

No - only fungible asset types are. We do not write an agreement that implements support for ERC721 NFTs, for example. Supporting NFTs in channel networks generally require assumptions/agreements about pricing the NFTs, how much the price can change over time, etc.

Does `expiry` mean the app instance has an expiry time?

Yes - this is a limitation of our current design. It exists so that the intermediary's capital is not locked up forever. The other way to guarantee this, and the general strategy to remove the expiry-time restriction, is to give the intermediary the ability to make an on-chain transaction to create a new direct channel, allowing them to recover their locked-up capital. However this is more complicated and is still in the "spec design" phase.

Can new app instances be installed without the intermediary's involvement?

No.

What topologies are supported?

Currently, a single intermediary connected in two direct channels to two different parties, who fund a virtual app with themselves as the signers.

The current roadmap for supporting more complex topologies is to remove the expiry-time restriction and at the same time reuse the existing interpreters infrastructure. This will also allow app instances to be installed without the intermediary's involvement.

CHAPTER 5

Protocols

A protocol is defined as the set of instructions executed and messages exchanged between a set of parties to achieve an outcome (e.g., installing a new state channel application). Instructions executed include producing digital signatures, reading persistent state to disk, etc.

Counterfactual writes client software that speak the same Counterfactual protocol, specified in the following page.

Many protocols are specialized to two-party and specified as such. We plan to extend them to n-party channels in the future.

5.1 Protocol Structure

A protocol consists of the following components:

- **Exchange**. The protocol exchange is the series of messages exchanged between all parties in the state channel, as well as dependencies between messages.
- **Message**. A message is the set of information that must be exchanged by the parties to recreate and validate the commitment signatures and associated transactions that those signatures enable. Each protocol may in general contain multiple message types.
- **Commitments**. A protocol produces one or more commitments. Both the signatures and the data contained within the commitments must be stored.

  Note: Messages are represented as JSON-encoded objects; the transport layer should be able to reliably send and receive such messages.

5.1.1 Commitment Structure

- **Signature**. The resultant data generated by computing a cryptographic signature over some hash representing the transaction to be executed.
- **Data**. Supplementary data that that allows any party in the protocol to reconstruct the hash and thus verify the signature.

- **Transaction Digest**. The transaction digest is the hash that is signed by each party, enabling the protocol's transaction to be executed on-chain. The calldata, if present, is used to generate the digest.
- **Transaction**. The transaction is the `(to, val, data, op)` tuple that a given protocol allows one to broadcast on-chain. These transactions enforce commitments created from the calldata and signature digests, manifesting the off-chain counterfactual state into the on-chain reality.

5.2 Primitive Types

5.3 JSON

This type specifies a modification of JSON that disallows the following primitive types: `true`, `false`, `null`. Note that when represented in javascript, large numbers which fit into `uint256` must be represented as hex strings, e.g., `0x01`.

Type: `CfAppInterface`

CHAPTER 6

Setup Protocol

6.1 Messages

After authentication and initializing a connection, channel establishment may begin. All state channels must run the Setup Protocol before any other protocol. As the name suggests, its purpose is to setup the counterfactual state such that later protocols can be executed correctly.

Specifically, the Setup Protocol exchanges a commitment allowing a particular off-chain application to withdraw funds from the multisignature wallet. We call this application instance the Free Balance application, representating the available funds for any new application to be installed into the state channel. The app definition is called IdentityApp.

Unlike other protocols, there is no extra message data for the Setup Protocol because the commitment digests are fully determined by the addresses of the participants.

6.1.1 The `Setup` Message

6.1.2 The `SetupAck` Message

6.2 Commitments

Commitment for `Setup` and `SetupAck`:

The commitment can be visually represented like:

CHAPTER 7

Install Protocol

The **Install Protocol** can be followed to allocate some funds inside of a `StateChannel` to a new `AppInstance`.

7.1 Commitments

In order for some of the funds in the multisignature wallet to be securely considered "allocated" to some `AppInstance`, there must exist a `DELEGATECALL` transaction to some contract which defines the logic for spending it and then executes the on-chain state transition to make it happen. In particular, this contract must do the following:

1. **Get the list of funded apps.** Go the `ChallengeRegistry` for the final outcome of the `FreeBalance` `AppInstance` to see the list of `activeApps`.
2. **Verify this app is funded.** Check that the `appInstanceIdentityHash` of a particular `AppInstance` is in the list of `activeApps`.
3. **Get the outcome of the app.** Go to the `ChallengeRegistry` for the final outcome of the `AppInstance` that these funds were allocated to
4. **Execute the effect.** Forward this outcome to the `interpreterAddress` along with some `interpreterParams` to have that interpreter execute the on-chain state transition.

So, the commitments involved in this protocol must do two things:

1. **Execute a conditional transaction.** Ensure that there is a `DELEGATECALL` to the above mentioned contract (i.e., the "delegate target")
2. **Update the free balance state.** Ensure that there is a signed message to update the `FreeBalance` `AppInstance`'s state to include the new `appInstanceIdentityHash` in the `activeApps` array with the right amount of funds decrement from `balances`.

7.1.1 The `ConditionalTransaction`

The pre-signed conditional transaction can be visually represented like this:

The digest that must be signed is the following:

```
keccak256(
  abi.encodePacked(
    byte(0x19),

    // Array of addresses of the owners of the multsig
    multisigOwners,

    // Address of the ConditionalTransactionDelegateTarget.sol library
    to,

    // A value of 0 as no ETH is being sent
    0,

    // The encoded function call to the delegate target
    abi.encodeWithSignature(
      // The name of the method in ConditionalTransactionDelegateTarget.sol
      "executeEffectOfInterpretedAppOutcome(address,bytes32,bytes32,address,bytes)",

      // Address of the global registry for on-chain challenges
      challengeRegistryAddress,

      // The unique identifier of the FreeBalance AppInstance
      freeBalanceAppIdentityHash,

      // The unique identifier of the AppInstance funds were allocated to
      appIdentityHash,

      // The address of the interpreter handling the outcome
      interpreterAddress,

      // Any extra data to pass to the interpreter function
      interpreterParams
    ),

    // An enum representing a DELEGATECALL
    1
  )
);
```

The signatures of this digest will be passed into the `execTransaction` method on the multisignature wallet.

7.1.2 The `FreeBalanceSetState`

The commitment to be signed to the `FreeBalance` `AppInstance` is identical to what would be signed in an **Update Protocol** commitment. The state update is such that the balances decrease of each participants in the `balances` array of their `FreeBalanceApp` and there is an added entry in the `activeApps` array.

7.2 Messages

7.2.1 Types

First we introduce a new type which we label `InstallParams`.

Type: `InstallParams`

NOTE: `participants` are deterministically generated based on the appSeqNo of the application in relation to the entire channel lifecycle. Specifically the key is computed as the (`appSeqNo`)-th derived child of an extended public key that is the unique identifier for a state channel user.

7.2.2 M1: Initiating signs `ConditionalTransaction`

7.2.3 M2: Responder countersigns `ConditionalTransaction` and signs `FreeBalanceSetState`

7.2.4 M3: Initiator signs `FreeBalanceSetState`

CHAPTER 8

Update Protocol

Once an application has been installed into the state channel, the multisignature wallet has transferred control over the installed amount from the free balance to the application's `computeOutcome` function, a mapping from application state to funds distribution. For example, in the case of Tic-Tac-Toe, a possible payout function is: if X wins, Alice gets 2 ETH, else if O wins Bob gets 2 ETH, else send 1 ETH to Alice and Bob.

As the underlying state of the application changes, the result of the payout function changes. It is the job of the Update Protocol to mutate this state, independently of the rest of the counterfactual structure.

Using our Tic-Tac-Toe example, if Alice decides to place an X on the board, Alice could run the Update Protocol, transitioning our state to what is represented by the figure above. Notice how both the board changes and the *local* versionNumber for the app is bumped from 0 to 1. To play out the game, we can continuously run the update protocol, making one move at a time.

8.1 Roles

Two users run the protocol. They are designated as `initiator` and `responder`.

8.2 Messages

For the below messages, the digest that is signed is represented as the following (reference: computeAppChallengeHash)

```
keccak256(
  ["bytes1", "bytes32", "uint256", "uint256", "bytes32"],
  [
    0x19,
    keccak256(                          //
      abi.encode(                       //
        [uint256, address[]],           // NOTE: This is the
        [channelNonce, participants]    // appInstanceIdentityHash
```

(continues on next page)

(continued from previous page)

```
      )                                    //
    ),                                     //
    0,
    TIMEOUT,
    appStateHash
  ]
);
```

Type: `UpdateParams`

8.2.1 The `SetState` Message

8.2.2 The `SetStateAck` Message

8.3 Commitments

Commitment for `SetState` and `SetStateAck`:

The commitment can be visually represented like:

This transaction invoke the `setState` function with the signatures exchanged during the protocol.

CHAPTER 9

Uninstall Protocol

The lifecycle of an application completes when it reaches some type of end or "terminal" state, at which point both parties know the finalized distribution of funds in the application-specific state channel.

In the case of a regular application specific state channel, both parties might broadcast the application on chain, wait the timeout period, and then broadcast the execution of the Conditional Transfer, thereby paying out the funds on chain. In the generalized state channel context however, the post-application protocol is to transfer the funds controlled by the application back to the Free Balance application off chain, so that they could be reused for other off-chain applications.

Using our Tic-Tac-Toe example, imagine Alice made the final winning move, declaring X the winner. If Alice runs the Uninstall Protocol, then the Counterfactual state transitions to what is shown above.

9.1 Messages

9.1.1 The `Uninstall` Message

9.1.2 The `UninstallAck` Message

9.2 Commitments

Commitment for `Uninstall` and `UninstallAck`:

There is one operation required for a successful uninstall.

- Set a new state on the Free Balance. The outcome function defined in the application must be run to compute an update to the Free Balance that is based on the outcome of the application.

Specifically, the Conditional Transfer commitment created by the Uninstall Protocol updates the free balance state with the app removed from the list of active apps and its outcome folded into the new state of the free balance.

CHAPTER 10

Install Virtual App Protocol

The **Install Virtual App Protocol** can be followed to allocate some funds inside of a `StateChannel` to a new `AppInstance` where the counterparty on the `AppInstance` does not have an existing on-chain multisignature wallet with funds inside of it. Instead, funds are "guaranteed" to the `AppInstance` via agreements on both ends of two `StateChannels` with an intermediary.

One way of thinking about this protocol is that it is essentially comprised of two Install Protocol executions. The first installation is between the initiator and the intermediary and the second between the intermediary and the responder. The installation produces `ConditionalTransaction` commitments which are signed but the distinction is that the `appIdentityHash` that is being pointed to is actually one where the `participants` are the initiator and the responder in both cases, and never the intermediary. Of course, then, the *interpreter* parameters that are used then become unique each installation and interpret the outcome of the "virtual app" differently. The purpose of the setup is to ensure that no matter the outcome, the intermediary will receive the same amount of funds so that they take on no risk.

10.1 Commitments

As mentioned above, there are `ConditionalTransaction` commitments that get signed which we label as a "VirtualAppAgreement". This agreement checks the outcome of the `AppInstance` being installed virtually and distributes the funds to the intermediary and the counterparty accordingly.

There is one additional commitment that is unique to this protocol, however, which is the `VirtualAppSetState` commitment. Since the `participants` of the `AppInstance` *include* the intermediary, we need a way of removing the requirement that every state must be signed by them. So, the `VirtualAppSetState` is exactly that; it commits on behalf of the intermediary that all state signed up until some `expiryVersionNumber` do not require the intermediary's signature to be valid.

10.1.1 VirtualAppSetState

There are two digests for this commitment. The first is for the initiator and responder parties to sign:

```
keccak256(
  abi.encodePacked(
    byte(0x19),
    identityHash, // The identity hash for the virtual app
    versionNumber, // The intial version number (will be 0)
    timeout, // The timeout for this state (will be the default)
    appStateHash // The hash of the initial state
  )
);
```

This is identical to any normal signed digest for a state update as you would see in the Update Protocol.

The second is for the intermediary to sign:

```
keccak256(
  abi.encodePacked(
    byte(0x19),
    identityHash, // The identity hash for the virtual app
    req.versionNumberExpiry, // Block up until which intermediary signature is not␣
↪required
    req.timeout, // The timeout for this state (will be default)
    byte(0x01)
  )
);
```

10.2 The `InstallVirtualAppParams` type

10.3 Messages

10.3.1 M1 - Initiator signs AB VirtualAppAgreement

10.3.2 M2 - Intermediary signs IB VirtualAppAgreement

10.3.3 M3 - Responding signs IB VirtualAppAgreement and IB FreeBalanceActivation

10.3.4 M4 - Intermediary signs AB VirtualAppAgreement and AI FreeBalanceActivation

10.3.5 M5 - Initiating signs AI FreeBalanceActivation and AB VirtualApp SetState

10.3.6 M6 - Intermediary signs IB FreeBalanceActivation and AB VirtualApp SetState

Note that in this message the intermediary is *forwarding* the initiator's signature on the AB VirtualApp SetState commitment.

10.3.7 M7 - Responding signs AB VirtualApp SetState

10.3.8 M8 - Intermediary sends initiator AB VirtualApp SetState

Note that in this message the intermediary is *forwarding* the responder's signature on the AB VirtualApp SetState commitment.

CHAPTER 11

Uninstall Virtual App Protocol

This is the Uninstall Virtual App Protocol.

11.1 Roles

Three users run the protocol. They are designated as `initiator`, `responder`, and `intermediary`. It is required that `initiator` and `responder` have run the `install-virtual-app` protocol previously with the same `intermediary`; however it is allowed to swap the roles of `initiator` and `responder`.

11.2 The `UninstallVirtualAppParams` type

At the end of this protocol the commitments `{left,right}ETHVirtualAppAgreement` defined in the `install-virtual-app` protocol are cancelled, and the free balances are updated.

11.3 Commitments

11.3.1 lockCommitment

The protocol produces a commitment to call `setState` with the final state of the `TimeLockedPassThrough` AppInstance in the `defaultOutcome` parameter and the `switchesOutcomeAt` value at `0`. This specific commitment is what guarantees that the "final outcome" of the virtual app is what is set inside of the `defaultOutcome` field; thereby "locking" it at this outcome.

11.3.2 uninstallLeft

A commitment to cancel the `leftETHVirtualAppAgreement` commitment produced by `install-virtual-app` and simultaneously update the free balance in the `initiator-intermediary`

free balance.

11.3.3 uninstallRight

A commitment to cancel the `rightETHVirtualAppAgreement` commitment produced by `install-virtual-app` and simultaneously update the free balance in the `intermediary-responder` free balance.

11.4 Signatures

11.5 Messages

11.5.1 M1

11.5.2 M2

11.5.3 M3

11.5.4 M4

11.5.5 M5

11.5.6 M6

11.5.7 M7

11.5.8 M8

CHAPTER 12

Withdraw Protocol

12.1 The `WithdrawParams` type

12.2 Commitments

12.2.1 installRefundApp

This is exactly the same kind of install commitment produced by the install protocol for regular apps. This commitment installs a balance refund app.

12.2.2 withdrawCommitment

This is a commitment for the multisig to send `amount` wei to `recipient`.

12.2.3 uninstallRefundApp

This is exactly the same kind of uninstall commitment produced by the uninstall protocol for regular apps and uninstalls the app installed by `installRefundApp`.

12.3 Signatures

12.4 Messages

CHAPTER 13

Glossary and Terminology Guide

For an introduction to the concepts and terminology behind state channels, please see the original paper.

13.1 State Deposit

Any kind of blockchain state controlled directly by a state channel. This could be an ETH balance, ownership of an ERC20 token, control over an ENS name registration, or any other kind of state.

13.2 State Deposit Holder

The on-chain multisignature wallet smart contract that is the "owner" of a given state deposit

13.3 Counterfactual Instantiation

The process by which parties in a state channel agree to be bound by the terms of some off-chain contract

13.4 Counterfactual Address

An identifier of a counterfactually instantiated contract, which can be deterministically computed from the code and the channel in which the contract is instantiated

13.5 Commitment

A signed transaction (piece of data) that allows the owner to perform a certain action

13.6 Action

A type of commitment; an action specifies a subset of transactions from the set of all possible transactions conditional transfer: the action of transferring part of the state deposit to a given address if a certain condition is true.

> NOTE: Section 6 of the paper specifies a concrete implementation that differs in certain respects from the protocol described here. The reason for this divergence is explained later.

CHAPTER 14

Contributing

HTML files are built automatically by readthedocs. To build a local preview copy, a Makefile is provided that invokes sphinx. Install the dependencies and build a local preview by doing the following:

```
python3 -m venv venv
source venv/bin/activate
pip3 install -r requirements.txt
make html
```