Channelising MixEth with the Counterfactual framework

Last week I visited the [L4](https://l4.ventures/) team in Canada with the intention of learning about the [Counterfactual](https://www.counterfactual.com/) framework they’re contributing to, and whether it is compatible with the [PISA watching service](http://hackingdistributed.com/2018/05/22/pisa/) I’m working on. The Counterfactual framework is still being developed, but enough of the smart contract layer is there to start designing a state channel application. So I decided to try to channelise an existing application and get some first hand experience with how the framework works.

Counterfactual is not the only state channel framework. I know of at least two more, [Magmo](https://magmo.com/) and [Celer](https://www.celer.network/), and I’m sure that there are others I don’t know about. This article is about the Counterfactual framework as it’s the only one I’ve had the chance to build an application with so far, but I look forward to trying out the others when I get a chance!

### MixEth

After getting some good ideas, [here](https://twitter.com/AFDudley0/status/1065422832249856000) and [here](https://twitter.com/Istvan_A_Seres/status/1055410823118684160), for useful channelised apps I eventually settled on adapting the [MixEth](https://github.com/seresistvanandras/MixEth) protocol. MixEth is a coin shuffling protocol for sending payments as part of an anonymity set. The aim is to hide individual senders and receivers. [This](https://www.youtube.com/watch?v=YgtF7psIKWg&feature=youtu.be&t=3701) excellent talk by Ian Miers highlights why decoys may not yield perfect privacy, but how they may provide some benefits over the transparency of Ethereum. MixEth’s main innovation is to achieve decoy protection with low gas costs — costs that are practical on the Ethereum we have today.



Harry Clark’s illustration of Edgar Allan Poe’s “Descent into the Maelstrom”. A fisherman witnesses the power of a mighty mixer and is never the same again ;)

In MixEth, senders deposit funds and name a public key whose private counterpart can unlock those funds. When enough deposits have been made to form a large enough anonymity set, participants — senders, receivers and anyone interested — are given the chance to perform a [verifiable Neff shuffle](http://web.cs.elte.hu/~rfid/p116-neff.pdf)on the public keys. After each shuffle any party can provide a proof that the shuffle was made incorrectly. When all interested parties have shuffled the the receivers can then withdraw their funds by providing proof of ownership of one of the shuffled keys.

The current gas costs incurred for a participant are 138,653 + 10,000\*n gas for a shuffle and 227,563 gas for a challenge, where n is the number of receivers/senders. These are the steps which we hope to remove by the channelising the app, thereby reducing the gas costs of the application. Channelising the app will also reduce the amount of time required on the part of each participant since state updates in channel are faster than those on-chain.

### Counterfactual

The Counterfactual framework aims to provide a full-stack solution for developers who wish to build state channel applications. My interests as a state channel researcher lie at the smart contract layer, and as such I am focused on this part of the framework. The framework allows developers to build an opinionated [AppDefinition](https://specs.counterfactual.com/02-state-machines#appdefinitions) which can then be leveraged by the rest of the framework to allow dynamic install and uninstall, and benefit from a standard challenge logic. Some simple examples can be found [here](https://github.com/counterfactual/monorepo/tree/master/packages/contracts/contracts/appDefinitions).



The Counterfactual stack: an off-chain application library, a generalised state channel protocol, and a set of on-chain contracts.

The documentation states that the AppDefinition should conform to an interface of the following four pure functions:

#### ****getTurnTaker: AppState => uint256****

This function accepts a state of the application and returns the index of the participant whose turn it is. The current method only allows one participant at a time to take a turn. This is a constraint that may be lifted at some point in the future.

getTurnTaker imposes the concept of turns upon an application. Applications that do not already have a strong idea of proceeding turn-by-turn will have to adopt this if they are to use the framework.

#### applyAction****: (AppState, Action) => AppState****

This is where the core of the application logic sits. applyAction accepts a state and an action and returns a new state. It typically entails a large switch statement where the developer maintains the logic that should be applied for a given action. The pure constraint on this function is something that developers should consider when adapting an application to the framework, application logic that references block.number or msg.sender will have to be adapted or removed. applyAction is also the only function that the framework allows to emit errors.

#### isStateTerminal: AppState => bool

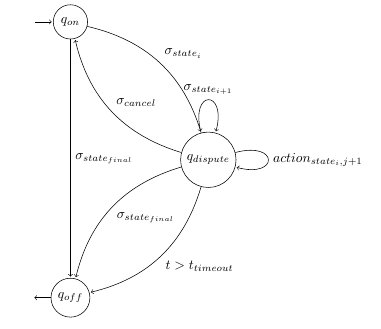
Here the framework introduces the notion of terminating an application. Examples would be a payment channel that would be terminal at any state, or a chess game that is only terminal after a win or draw can be declared. Applications that do not already have the notion of a lifecyle, or termination, will have to break up their app into terminal segments.

#### resolve: AppState => Transfer.Transaction

Decides what resolution the application should be for a given state. Developers should be aware that resolve could be called at any time, not just at a terminal stage, and so should provide a resolution for any possible state that the application could be in.

These functions are leveraged by the framework to allow participants to verify what the application would do were it to be deployed on-chain. If this were to be the case the challenge logic defined in [AppInstance](https://specs.counterfactual.com/02-state-machines#appinstance) would be used to progress the application.

Counterfactual’s challenge logic is as follows: To begin a challenge a participant needs to provide a state signed by all channel participants. From there the next turn taker can progress the application by applying an action. Subsequent participants can progress the application in the same way. If at any time the application transitions to a terminal state, resolution logic will be executed. Also, at any time another participant my submit a newer state than the one originally provided. This challenge process can also be exited and the application continued off-chain, if at any time all participants sign an order to cancel the dispute.



A state diagram of the Counterfactual challenge logic implemented in AppInstance. The channel transitions from states ‘**on**’ to ‘dispute’ when the a co-signed state is submitted. From there state is advanced by supplying actions. The channel can transition back to on if a cancel is signed by all participants. Or to off, if a final state is reached or a timeout occurs.

### Channelising MixEth

#### Some points to consider

MixEth has n number of participants, where n would typically be in double digits. Although I understand the front-end and communication layers of the Counterfactual stack are initially being designed for 2-party interactions it is the intention that the framework eventually support n-party applications. The smart contract layer has been written to allow for this, so MixEth should be a good test of how an n-party channel app could work within the framework.

The participants in MixEth have different interests at different stages of the protocol. In the on-chain version a sender submits funds, and optionally shuffles, after that their involvement is not required to complete the application. In state channels all participants must sign state updates, which means the senders will need to be involved until the end (and likewise the receivers from the beginning). Will the turn taking constraint adapt well to this, or will it force senders to take turns even when non are required?

MixEth has a timing assumption baked into the protocol. Participants are given time after each shuffle to submit a challenge. Given that these timings are measured with block.number, how can this be handled by the pure applyAction function?

The on-chain only application is a monolith, in that a single contract is used globally for all mixing. How can this be adapted for a fixed set of participants, and how can a terminal state be imposed to allow these participants to exit?

### Observations and discussion

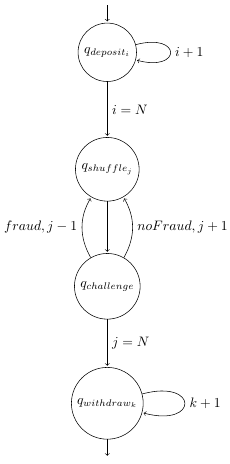
#### Overall

The original MixEth can be found [here](https://github.com/seresistvanandras/MixEth/blob/master/contracts/MixEth.sol), and the one converted to the Counterfactual framework [here](https://github.com/seresistvanandras/MixEth/blob/master/contracts/MixEthChannel.sol). The channelised contract doesn’t have any tests yet — the main purpose of this exercise was to understand **how** an application could be converted. All of the MixEth logic is mapped into the applyAction function. All storage is removed and modelled as a single struct — AppState. And all function arguments are merged into a single Action struct. Below are some details and discussion about the process.

#### Happy case terminal state

Since both senders and receivers need to make active actions in MixEth, both of these sets will need to be participants in the state channel. The next step is to identify a terminal state. In the on-chain MixEth, a sender completes when they have deposited funds, and optionally taken a shuffle. A receiver is finished when they have taken a shuffle and provided a proof of withdrawal signature. Since the channel contains both types of participants, and sets of each, the application should consider a state terminal when all participants have completed their respective tasks ie. when all senders have deposited, all shuffling has taken place, and all receivers have withdrawn.

#### A fixed participant state diagram



A state diagram of MixEth with the fixed number of participants. Each sender deposits funds. Then a shuffle occurs, followed by a challenge. A correct shuffle proceeds to the next shuffle, an incorrect shuffle reverts to the last shuffle. When all shuffles have taken place, each of the receivers withdraws funds.

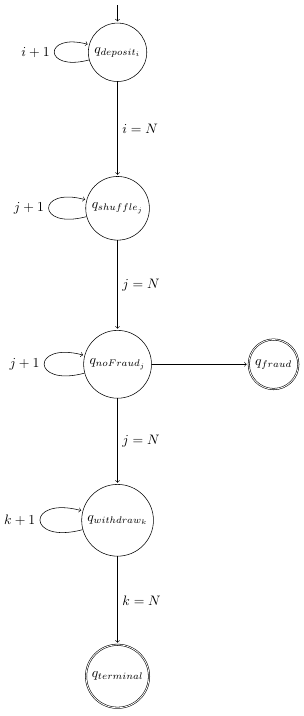
State machine diagrams are a useful way to visualise the possible transitions in smart contract applications, they’re also a nice way to inspect the flow for mistakes and stuck states.

This state machine diagram represents the transitions for MixEth with a fixed set of participants. A state diagram also helps us to recognise a sensible order on turns. The senders each deposit and name a key, then shuffles occur, after each shuffle there is the chance to challenge, and finally each receiver withdraws.

#### Time based constraints

In the original MixEth a challenge is allowed after each shuffle. This is achieved by allowing a time after the shuffle during which anyone can submit a challenge. To do this the Counterfactual framework would need to allow the concept of time to be used in applyAction. Theoretically it is possible to downgrade the pure constraint to view for this function, but this makes agreement between clients more complex — L4 are still working on this process. getTurnTaker would also have to be able to return a list of participants as anybody could validly take a turn at this point, which is also not currently possible in the framework. Another way that this functionality can be achieved is to explicitly allow each participant a turn to declare whether or not they would like to challenge. If they wish to challenge they submit the relevant data, if they don’t they sign that they wish to pass their turn. After allowing each party to take a turn like this it is possible to move on to the next shuffle, safe in the knowledge that no-one wishes to challenge.

#### Mitigating O(n²) complexity



A state diagram of the final MixEth with a single challenge round after all shuffles. Each sender deposits funds. Then all shuffles take place. After all shuffles take place each participant is given the chance to challenge any shuffle. If a shuffle is proved incorrect the application transitions to fraud, and all participants except the one at fault are refunded. If all shuffles are accepted each of the receivers submits withdrawal proofs. After this the application transitions to a final state and distributes funds.

[This article](https://medium.com/@stonecoldpat/you-sank-my-battleship-7cfdc4533bf5) shows that, in the worst case, state channel applications will have to be executed on-chain. Having an explicit move for each participant after each shuffle means that the application would scale at O(n²), for n participants.

To reduce the possible number of challenge rounds the state transitions were adjusted to not have a challenge after each shuffle. Instead, after all shuffles have taken place each participant is given the opportunity to challenge **any**shuffle. Now the application state will have to store all the shuffles — not just the last two — but the number of transactions would scale as O(n) for the application (and O(1) for each participant). This also gives rise to another terminal state — fraud. The resolve functionality for the fraud state is to revoke the deposit of the cheater, and refund all other participants.

#### Msg.sender

The original contract had references to msg.sender but this was just to ensure that the sender had the rights to call that function at that time — a point that would now simply be handled by the framework and the getTurnTakerfunction.

#### Application state

The whole application state is injected to applyAction, which for applications like MixEth with a large state, could be gas intensive. These costs can be mitigated by storing hashes of certain sub-sections of the state and proving the presence of the full data only when it needs to be used — an optimisation I’ve yet to make. An example of this would be store a hash of each shuffle round — rather than the full shuffle which is an array of size n — then a challenger would provide this data at challenge time, instead of each subsequent action.

Passing a struct to applyAction requires the use of the experimental ABIEncoderV2, which doesn’t allow for mappings to be included in the struct. Given that the original MixEth contract held almost all of its state in mappings, this constituted a significant part of the refactoring effort as the underlying data storage changed structure.

#### Setup constraints

The original MixEth marks a deposit function with the payable modifier, and this is how senders add ETH to the mix. It also checks that the value deposited is correct — in MixEth all participants must deposit the same amount so that an external party can’t use the amounts to infer who a sender has sent to. In the Counterfactual framework deposits are made to a multisig, then an application is installed and funds are allocated to it. Counterfactual makes balance checks occur as part of the [install sub-routine](https://specs.counterfactual.com/05-install-protocol). For MixEth, the default balance check that all participants have supplied the same amount is suitable. Applications requiring more complex logic would need to write a custom module that would be injected in the install sub-routine as middleware.

### Summary

With these points taken into consideration it’s possible to channelise the MixEth contract. In the original MixEth a transaction is required for each of a deposit, a shuffle, a possible challenge and a withdrawal. In the channelised version a participant only has to send ETH to a multisig wallet, then follow the Counterfactual off-chain protocol to install the shuffle app and sign state updates. Once this is done, a final transaction can be submitted to withdraw funds. **The expensive shuffle and challenge phases are no longer necessary on-chain.**

There are challenges in learning the Counterfactual framework and adapting an application to meet its constraints. But the constraints are in place to help **avoid some of the pitfalls of building a state channel app**. In writing the application I didn’t have to consider a safe protocol for allowing users to deposit funds, and still get refunded if another party didn’t respond in the setup phase. I also didn’t haveto writeany ofmy own challenge logic for when cooperation breaks down in a channel.

Developers building with the framework will **benefit from other complex components in the stack** — the communication layer between participants, the off-chain installation that allows end users to install any application without an on-chain transaction, the engine that monitors for blockchain events and compatibility with other applications built on the Counterfactual stack.

Many thanks to [Liam Horne](https://github.com/snario) from L4 for help in adapting the application to the Counterfactual framework and to [István András Seres](https://github.com/seresistvanandras) for helping me to understand the MixEth protocol.