Shell Protocol Description

April 27, 2020

1 High level description

The goal of the Cowri Shell Protocol is to unify stablecoins into a coherent monetary system. This current release is a first step towards that vision. It is a simple liquidity pool and automated market maker (AMM) for stablecoin-to-stablecoin trades. The pool has three primary behaviors:

1. Liquidity providers can deposit tokens into the pool
2. Liquidity providers can withdraw tokens from the pool
3. The pool can autonomously swap tokens using its deposited liquidity

2 Utility function and shell tokens

Formally, a pool can be thought of as an economic agent with a portfolio of tokens $\mathbf{x} = (x_1,\ x_2,\ ..., x_n)$, and a utility function $U(\mathbf{x})$ that assigns a scalar value to any given portfolio. A pool also issues its own fungible, ERC-20 compliant token, *shells*, but we will cover that later. The utility function is composed of a gain function, $G(\mathbf{x})$, and a loss function, $F(\mathbf{x})$:

$$U(\mathbf{x}) = G(\mathbf{x}) - F(\mathbf{x}) \qquad G(\mathbf{x}) = \sum_{i=1}^{n} x_i \qquad F(\mathbf{x}) = \sum_{i=1}^{n} f(x_i)$$

The intent of the loss function, F, is to penalize a pool once it crossed a certain threshold and is no longer considered, "balanced". A balanced pool is one where each token, x_i, is aligned with its pre-set allotment, $weight_i$. The total loss, F, is the sum of each token's individual loss, $f(x_i)$, which can be described as follows:

$$f(x_i) = \begin{cases} 0, & \beta_L \leq x_i \leq \beta_U \\ (x_i - \beta_U)^2 * \delta / ideal_i, & x_i > \beta_U \\ (\beta_L - x_i)^2 * \delta / ideal_i, & x_i < \beta_L \end{cases}$$

$$ideal_i = weight_i * \sum x_i \qquad \beta_U = ideal_i * (1 - \beta) \qquad \beta_L = ideal_i * (1 + \beta)$$

2.1 Minting and burning shells

Whenever a liquidity provider deposits tokens, the pool mints *shells*, an ERC-20 compliant token. To withdraw tokens from the pool, the liquidity provider burns *shells*. The income from trading fees (see Section 3) will automatically accrue to the holders of *shells*. Whether depositing or withdrawing, the percent change in the utility function will equal the percent change in the supply of shells (assuming there are no fees). We can describe this relationship formally based on a transition from an initial portfolio $\mathbf{x_0}$, to a subsequent portfolio, $\mathbf{x_1}$:

$$\frac{U(\mathbf{x_1}) - U(\mathbf{x_0})}{U(\mathbf{x_0})} = \frac{S_0 - S_1}{S_0}$$

$$\frac{G(\mathbf{x_1}) - F(\mathbf{x_1}) - G(\mathbf{x_0}) + F(\mathbf{x_0})}{U(\mathbf{x_0})} = \frac{S_0 - S_1}{S_0}$$

$$\Delta S = \frac{\Delta G - \Delta F}{U_0} * S_0$$

The above equation still requires one more modification. The intent of the loss function, F, is to penalize an unbalance pool vs. a balanced pool. When a transaction makes a pool more unbalanced, i.e. $\Delta F > 0$, and there will be slippage. For a deposit, that means fewer shells will be minted. For a withdrawal, more shells will be burned. For a swap, the price will be less favorable for the trader. When a transaction makes a pool less unbalanced, $\Delta F < 0$, then there will be anti-slippage. One of the design goals for this implementation of the Shell Protocol is to offer dynamic liquidity provider fees. Most if not all existing liquidity pools give all of the anti-slippage to arbitrage traders, people who rebalance the pool. We want some of the anti-slippage to go to liquidity providers as a dynamic fee. We can accomplish this by adding a modification to the above equation: if $\Delta F < 0$, we multiply the loss function by a new parameter, $0 \leq \lambda \leq 1$:

$$\Delta S = \frac{\Delta G - \lambda \Delta F}{U_0} * S_0$$

With this modification added when a pool becomes more balanced, we ensure that liquidity providers will ultimately earn $\lambda * 100\%$ of the slippage from transactions that unbalance the pool, even after the pool has been rebalanced.

3 Halt checks, tuneable parameters and fixed fees

Before we describe and formulate logic for the actual transactions, it is worth explaining a few miscellaneous concepts. First, are the halt checks. We want to ensure that when a stablecoin breaks its peg, the entire pool will not be drained. If this happens, then liquidity providers would lose all of their money, even if the broken coin was only 10% of the total by weight. Without a halt check ensuring the pool cannot be drained of all value, then the pool will become more risky for liquidity providers the more stablecoins added, the opposite of risk diversification. In order for a transaction to be valid, a halt check requires that the token balance at the end of a transaction is within a certain threshold:

$$ideal_i * (1 - \alpha) \leq x_i \leq ideal_i * (1 + \alpha)$$

This is probably a good opportunity to explain the role of each of the five tuneable parameters set by the contract owner, such as α:

$0 < \alpha < 1$: halt boundary

$0 < \beta < 1$: slippage boundary

$0 < \delta$: rate of increase for slippage calculation

$0 < \epsilon < 1$: fixed fee

$0 < \lambda < 1$: liquidity provider share of anti-slippage

It is worth going into more detail regarding the application of the fixed fee, ϵ. Any time tokens enter the pool or leave the pool, the fixed fee is applied. So for a deposit, the fee is applied once, the same for a withdrawal. But for a swap, tokens simultaneously enter and leave the pool, so the fixed fee must be applied twice.

4 Adding and removing liquidity

There are two ways to add and remove tokens from the pool, "selective" and "proportional." For selective deposits and withdrawals, the user specifies exactly how many of each token to add or remove. The pool then calculates how many shell tokens to mint or burn. For proportional deposits and withdrwals, the tokens are added and removed in relative amounts that are proportional to what is already in the pool.

4.1 Selective deposits and selective withdrawals

For selective deposits, the user inputs how many of each token they want to add to the pool, $\mathbf{d} = (d_1,\ d_2,\ ...,\ d_n)$. To calculate how many shells will be minted, ΔS, we use the following equation:

$$\Delta S = \frac{U(\mathbf{x}+\mathbf{d}) - U(\mathbf{x})}{U(\mathbf{x})} * S * (1-\epsilon)$$

$$\Delta S = \frac{\sum d_i - \Delta F}{U(\mathbf{x})} * S * (1-\epsilon) \qquad \text{OR} \qquad \Delta S = \frac{\sum d_i - \lambda \Delta F}{U(\mathbf{x})} * S * (1-\epsilon)$$

We can easily extend this equation to apply to selective withdrawals, where the user specifies how many of each token to take out of the pool, $\mathbf{w} = (w_1,\ w_2,\ ...,\ w_n)$ and the protocol calculates how many shells to burn, ΔS:

$$\Delta S = \frac{U(\mathbf{x}) - U(\mathbf{x}-\mathbf{w})}{U(\mathbf{x})} * S * (1+\epsilon)$$

$$\Delta S = \frac{\sum w_i - \Delta F}{U(\mathbf{x})} * S * (1+\epsilon) \qquad \text{OR} \qquad \Delta S = \frac{\sum w_i - \lambda \Delta F}{U(\mathbf{x})} * S * (1+\epsilon)$$

4.2 Proportional deposits and proportional withdrawals

For proportional deposits, the user specifies how many tokens they want to withdraw in total, $\sum d_i$, and the protocol then calculates two things: how many of each individual token to deposit (d_i) and how many shells to mint (ΔS). By design, a proportional deposit should not create any slippage because the ratios of tokens before and after the transaction do not change, and hence the pool does not become more or less unbalanced. Here are the equations:

$$\Delta S = \frac{\sum d_i}{\sum x_i} * S * (1-\epsilon) \qquad d_i = \frac{x_i}{\sum x_i} * \sum d_i$$

For proportional withdrawals, the user specifies how many shell tokens to burn, ΔS, and the pool calculates how many of each token to withdraw, w_i. The withdrawal amounts will have the same ratio to each other as the token balances in the pool. Here is the equation to calculate the withdrawal amounts:

$$w_i = \frac{\Delta S}{S} * x_i * (1 - \epsilon)$$

5 Swapping tokens

A swap, where a user trades with the pool to exchange one token for another, can be thought of as an atomic deposit and withdrawal such that the pool's utility does not change before or after. The user deposits token d_i and withdraws token w_j in amounts such that $\Delta U = 0$.

$$U(\mathbf{x}) = U(\mathbf{x} + d_i - w_j)$$

Using this equality as a starting point, we can simplify the relationship between d_i and w_j as follows:

$$w_j = d_i + F(\mathbf{x}) - F(\mathbf{x} + d_i - w_j) \qquad \text{OR} \qquad w_j = d_i + \lambda * [F(\mathbf{x}) - F(\mathbf{x} + d_i - w_j)]$$

Unfortunately, these equations do not have a simple analytic solution because F is a quadratic function that takes as an input the variables we are trying to solve for. Therefore, we devised an algorithmic approach to iteratively converge on an estimate. The following subsections explain how this approach works for both origin swaps (user inputs d_i) and target swaps (user inputs w_j).

5.1 Origin swaps

In an origin swap, the user provides the amount of origin tokens they are willing to trade, d_i, and the protocol calculates how much of the target token to swap, w_j. We execute the following algorithm until it reaches convergence:

$$w_0 = d_i$$

$$w_{t+1} = d_i * (1 - \epsilon) + F(\mathbf{x}) - F(\mathbf{x} + d_i * (1 - \epsilon) - w_t)$$

OR

$$w_{t+1} = d_i * (1 - \epsilon) + \lambda * [F(\mathbf{x}) - F(\mathbf{x} + d_i * (1 - \epsilon) - w_t)]$$

Upon completion, we assess a final fee on the target amount: $w_i = w_t * (1 - \epsilon)$

5.2 Target swaps

Target swaps are just like origin swaps, except the user specifies how many tokens they want to buy, w_j, and the smart contract calculates how many tokens the user must sell, d_i. We execute the following algorithm until it reaches convergence:

$$d_0 = w_j$$

$$d_{t+1} = w_j * (1 + \epsilon) - F(\mathbf{x}) + F(\mathbf{x} + w_j * (1 + \epsilon) + d_t)$$

OR

$$d_{t+1} = w_j * (1 + \epsilon) - \lambda * [F(\mathbf{x}) - F(\mathbf{x} + w_j * (1 + \epsilon) + d_t)]$$

Upon completion, we assess a final fee on the origin amount: $d_i = d_t * (1 + \epsilon)$