Recurring Payments Smart Contracts

Fabrice Le Fessant
OCamlPro SAS
Telegram: @fabrice_dune
https://github.com/OCamlPro/devex-27-recurring-payments

September 1, 2021

1 Executive Summary

This document describes a Recurring Payments System developed by Fabrice LE FESSANT for FreeTON. The system allows *Service Providers* to publish *Services*, with a *cost per period*. Then, *Service Users* can subscribe to such services for a predefined number of periods, paying for the whole period. However, Service Providers can only spend the tokens for the current period and already elapsed periods. Users can *stop* a subscription (receiving back the tokens for the periods that will not be consumed, or they can just *pause* the subscription (they can later *unpause* such subscriptions).

The main property is that, to subscribe for a large set of periods, only one transfer is needed. Another transfer will only happen if the user decides to stop the subscription, and be refunded.

The System supports both TON tokens and TIP-3 tokens for all services (a given Service must choose only one currency to use, but a Service Provider can publish Services using different currencies). The current System uses Broxus TIP-3 tokens.

Service users and service providers can monitor their contracts for events, corresponding to changes in the subscriptions, or provide a callback, i.e. a contract that will receive a message every time a change is done on subscriptions.

The System has been deployed on the Testnet, and debots are available for Service Providers and Service Users. A set of test scripts is also available to test on TONOS-SE.

Contents

2 Source code

The source code of all smart contracts is available at https://github.com/OCamlPro/devex-27-recurring-payments.

All smart contracts are written in Solidity, preprocessed with `cpp -E` and `ft` substitutions.

3 Deployment

The system has been deployed on the testnet.

The following addresses can be used:

Root contract: `0:36b2657ba8546a1a7bdd06c9ce9062c7e998585a05fc00c03f57dc75546c96c4`
The `RPSRoot` contract shared by all contracts of a given System;

Provider debot: `0:af6e90a43565598f7b59a283416bacc2bebb222064c9b4d309b8e6725500fd6e`
This debot can be used to act as a Service Provider, i.e. deploy a `RPSProvider` contract, list services, add services and claim subscriptions for elapsed periods;

User debot: `0:ca8d11ffa44009a02ce4b02e42631e5c8c96042f1c0aae57acae132b29ed13a4`
This debot can be used to act as a Service User, i.e deploy a `RPSUser` contract, check subscriptions, subscribe to services and manage them (stop, pause and unpause);

Provider contract: `0:bd6c0225544eac8f4314e33d88b7e252cf6d5e002cf77654eeb4a03e11c5ec7f`
This is an example Service Provider contract with only one service, with a period of 1 minute, and a cost of 1 ton per minute;

These contracts have been deployed with TVC images corresponding to GIT commit `0ae44f292dab59f790a9efd618a5029a2e4333fb`.

4 Architecture

4.1 Smart contracts

The System provides 3 different kinds of contracts:

`RPSRoot` contract: the `RPSRoot` contract is in charge of deploying smart contracts for Service Providers and Service Users.

The code is available here: https://github.com/OCamlPro/devex-27-recurring-payments/blob/master/contracts/RPSRoot.spp

The contrat is used as follows:

- After deployment, the owner must call functions `setProviderCode` and `setUserCode` to provide the code of other contracts.

- Before deploying a contract, a user must first provide some funds to the contract. This is done by calling the `creditBalance` function, to associate the funds with the user's pubkey. Once his balance contains enough TONs, the user can either call `deployProvider` or `deployUser` functions.
- The contract also provides utility functions, such as `getUserAddress` and `getProviderAddress` to compute addresses from pubkeys.

`RPSProvider` contract: the `RPSProvider` contract is in charge of managing the services and subscriptions for a given Service Provider (identified either by his pubkey or his address).

The code is available here: https://github.com/OCamlPro/devex-27-recurring-payments/blob/master/contracts/RPSProvider.spp

This contract is used as follows:

- To manage Services, the contract provides two functions `addService` and `getServices`. Services have a `name`, a `description`, a `period` (in seconds) and a `period_cost` (in nanotons). Once added, services receive a uniq identifier for the provider, called `serv_id`.
- The contract provides different entry points that can only be called by `RPSUser` contracts. They are used to get information on services, subscribe to services, and then stop, pause and unpause subscriptions.
- Finally, the Service Provider can claim the tokens that can already be spent: they correspond to elapsed periods and already started periods. Tokens for periods in the future are locked and cannot be used, so that the user can be refunded if he stops the subscription. To claim tokens, the Service Provider must first call `claimSubscriptions` to unlock tokens for all subscriptions. Once tokens have been unlocked, the Service Provider can call the `transferClaimed` function, providing the token root (address 0 is used for TONs) and the destination wallet.

`RPSUser` contract: the `RPSUser` contract is in charge of managing subscriptions for a given user (identified either by a pubkey or an address).

The code is available here: https://github.com/OCamlPro/devex-27-recurring-payments/blob/master/contracts/RPSUser.spp

The contract is used as follows:

- The user must transfer tokens to the contract to be able to subscribe to services. For that, the user may use `needWallet` to create a wallet for a specific TIP-3 token, and `getWallet` to get the address of the wallet, and its current balance. He can then transfer tokens to the wallet. He can also use `transfer` to recover some of the tokens to his personal wallet.
- The user can use `subscribe` to subscribe to a particular service. He must provide `provider`, the address of the Service Provider contract, `serv_id`, the identifier of the service, `periods`, the number of periods, and `callback`, the address of a callback contract (or the address

0). The user can use `getSubscriptions` to check if the service was correctly added. Events and callbacks are used to warn the user if a problem occurred and the service could not be subscribed.

- For every subscription, the user can call `stopSubscribe`, `pauseSubscribe` and `unpauseSubscribe`. `stopSubscribe` will refund the user for periods that have not yet started. `pauseSubscribe` pauses the subscription, and `unpause` continues a paused subscription, with exactly the same remaining time as when it was paused.

4.2 Debots

The System provides 2 different kinds of debots:

`RPSUserDebot`: This debot allows a user to manage his subscriptions. It must be initialized with the address of the root contract. If the `RPSUser` contract of the user has not yet been deployed, the debot will transfer an initial credit from the user's multisig to the root contract, and then deploy the contract.

The code is available here: https://github.com/OCamlPro/devex-27-recurring-payments/blob/master/debot/RPSUserDebot.spp

`RPSProviderDebot`: This debot allows a Service Provider to manage his services and claim his subscriptions. It must be initialized with the address of the root contract. If the `RPSProvider` contract has not yet been deployed, the debot will transfer an initial credit from the owner's multisig to the root contract, and then deploy the contract.

The code is available here: https://github.com/OCamlPro/devex-27-recurring-payments/blob/master/debot/RPSProviderDebot.spp

4.3 Multiple Currencies

Both `RPSUser` and `RPSProvider` inherits from `MultiWallet` contract, a contract that is used to manage multiple wallets for multiple TIP-3 tokens. The address 0 is reserved for TONs.

The code is available here: https://github.com/OCamlPro/devex-27-recurring-payments/blob/master/contract/MultiWallet.spp

4.4 Development Tools

We provide debots so that the contracts can be easily tested. For the development itself, we used `ft`, https://github.com/OCamlPro/freeton_wallet, both to build Solidity contracts, deploy them and test them.

During the contest, many new features were added to `ft`, especially to ease the development of debots. In particular, Solidity files with `spp` extension are preprocessed using `cpp -E`, and we provide a set of macros to simplify the development of depots. https://github.com/OCamlPro/devex-27-recurring-payments/blob/master/debot/lib/cpp.sol