UCP CBDC Docs

Copyright 2022, Reserve Bank of Australia

Commercial-in-Confidence

Early Access Draft for Review

Overview

This documentation gives a quick guide to connecting and interacting with the CBDC ledger for use case providers (UCP). The CBDC is managed as a smart contract running on a private permissioned blockchain. The smart contract can be accessed using a URL associated with a specific “app cred”. Each UCP will be provided with two app creds intended to be used as follows:

- UCP credential: to be used only by the UCP for functions they use directly
- UCP-User credential: to be shared with, and used by, all of the End Users for each UCP if required, e.g. for Users to manage self-custodial holdings of CBDC, or independently validate balances or transaction receipts.

If app creds are made public or otherwise abused in the project, they may be revoked. Reissued app creds would require client integrations to be reconfigured to use the new app creds.

Along with this documentation, we also provide some sample code as a demo application illustrating how to integrate with the CBDC system.

Supported Privacy Models

Please note that the CBDC system currently only supports a **semi-private ledger** model.

Semi-private ledger - verifiable by Pilot participants only using issued app creds.

Connecting Using Fireblocks

For connections using Fireblocks, please reach out to the DFCRC team for alternate connection details.

Connecting To The Network

A subset of the Ethereum JSON-RPC interface is made available for UCPs and their End Users. Refer to Ethereum documentation on JSON-RPC for general information on the use of this API, such as available at the following link.

https://documenter.getpostman.com/view/4117254/ethereum-json-rpc/RVu7CT5J#e15b7abf-6207-a225-0d64-3d33cab980cb

The CBDC smart contract can be accessed by using the URL associated with each app cred. The URL contains a shared username and password, embedded in the URL structure as follows:

```
1https://{username}:{password}@a0jq79osep-a0yjwjns56-rpc.au0-aws.kaleido.io
```

The address of the CBDC smart contract is provided below.

```
10xb82C4150d953fcCcE42d7D53246B5553016c5C71
```

Connecting To Metamask Web Extension

The following section describes how to connect a browser-based Metamask wallet to the CBDC system. We expect that similar steps can be performed to configure other wallets to access the CBDC system.

- Click on “Add Network”
- This should open the following page
- Fill in the page using the following information
  - Network Name: CBDC
  - New RPC URL: `https://{username}:{password}@a0jq79osep-a0yjwjns56-rpc.au0-aws.kaleido.io/`
  - Chain ID: 1007845588
  - Currency Symbol: CBDC
  - Block Explorer URL: Leave blank
- You then need to import the tokens into Metamask.

    - Click import tokens

Insert the following information

- Token Contract Address: 0xb82C4150d953fcCcE42d7D53246B5553016c5C71

Token Symbol and Token Decimal should be filled automatically.

JSON RPC Limitation

The CBDC system uses a blockchain application firewall; the full JSON-RPC interface is not made available to UCPs and Users. Note that Gas and Uncle blocks are not applicable to the CBDC system. The following endpoints are available and applicable.

- `debug_traceBlockByNumber`
- `debug_traceTransaction`
- `eth_accounts`
- `eth_blockNumber`
- `eth_call`
- `eth_chainId`
- `eth_coinbase`
- `eth_gasPrice`
- `eth_getBalance`
- `eth_getBlockByHash`
- `eth_getBlockByHash`
- `eth_getBlockByNumber`
- `eth_getBlockTransactionCountByHash`
- `eth_getBlockTransactionCountByNumber`
- `eth_getCode`
- `eth_getFilterChanges`
- `eth_getFilterLogs`
- `eth_getLogs`
- `eth_getStorageAt`

- `eth_getTransactionByBlockHashAndIndex`
- `eth_getTransactionByBlockNumberAndIndex`
- `eth_getTransactionByHash`
- `eth_getTransactionCount`
- `eth_getTransactionReceipt`
- `eth_getUncleByBlockHashAndIndex`
- `eth_getUncleByBlockNumberAndIndex`
- `eth_newBlockFilter`
- `eth_newFilter`
- `eth_newPendingTransactionFilter`
- `eth_protocolVersion`
- `eth_sendRawTransaction`
- `eth_subscribe`
- `eth_uninstallFilter`
- `net_version`
- `txpool_content`
- `web3_clientVersion`

Note that:

- Smart contract deployment is blocked. That is, UCPs and Users cannot deploy smart contracts onto the CBDC system.
- Calls to eth_call can only be directed to the provided CBDC contract address. Calls to any other contract address will be blocked.
- `eth_sendTransaction` is blocked as managed signing is not allowed.

KYC

Before any address can send or receive CBDC, the address needs to be KYCed by an authorized KYCer, and be marked as being KYCed on the CBDC system. Identity information is not stored on the CBDC ledger, but rather is kept separately by the

authorized KYCer entity. It is important to note that only signing addresses should be subject to KYC verification.

It is also possible for an address to be KYCed by multiple KYCers. In such cases, losing KYC status with one KYCer does not prevent the address from continuing to engage in CBDC transactions. To restrict an address from conducting CBDC operations, it must lose KYC status with all of the KYCers that have granted it.

KYCer

Certain UCPs may also be designated as KYC verifiers or KYCers. To be approved as a KYCer, a UCP must first obtain permission from the Reserve Bank of Australia (RBA). Upon approval, the RBA will grant the KYCer role to the signing address of the UCP.

Once granted the KYCer role, the address will have the ability to use the "grantKYC" and "revokeKYC" functions to mark and remove the KYC status of other addresses in the CBDC system.

Freezing

The RBA has the authority to freeze any address in the CBDC system, which prevents the address from engaging in transactions on the network. When an address is frozen, a "Frozen" event will be emitted to the CBDC ledger. It is important to note that while the Frozen address is restricted from transacting, the rest of the network will continue to operate normally. The address may be unfrozen by the RBA at a later time, at which point an "Unfrozen" event will be emitted to the ledger. Once unfrozen, the address will be able to resume transactions on the CBDC network.

Pausing

The RBA has the authority to pause the CBDC network, which temporarily suspends all operations on the network. While the system is paused, no addresses will be able to conduct transactions. When the network is paused, a "Paused" event will be emitted to the CBDC ledger. At a later time, the RBA may choose to resume normal operations on the network by unpausing it. When the network is unpaused, an "Unpaused" event will be emitted to the ledger, and all addresses will regain the ability to transact.

CBDC Smart Contract Interface

The source code in the example interface files include only the latest event names. In the Sandbox environment, some events were named differently in previous versions of the smart contract. If you require these old event names, please contact the DFCRC's CBDC Tech team.

The following contains the interface for Users (UCP and UCP-USERS) to interact with the platform.

IUser

This interface implements ICheckKYC, IERC20Custom, IHTLC, IRenounceable. More information on these interfaces are provided below.

totalSupply

```
1function totalSupply() external view returns (uint256);
```

Returns the amount of tokens in existence.

balanceOf

```
1function balanceOf(address account) external view returns (uint256);
```

Returns the amount of tokens owned by `account`.

transfer

```
1function transfer(address to, uint256 amount) external returns (bool);
```

Moves `amount` tokens from the caller's account to `to`. Returns a boolean value indicating whether the operation succeeded.

allowance

```
1function allowance(address owner, address spender) external view returns (uint256);
```

Returns the remaining number of tokens that `spender` will be allowed to spend on behalf of `owner` through {transferFrom}. This is zero by default. This value changes when {approve} or {transferFrom} are called.

approve

```
1function approve(address spender, uint256 amount) external returns (bool);
```

Sets `amount` as the allowance of `spender` over the caller's tokens. Returns a boolean value indicating whether the operation succeeded. IMPORTANT: Beware that changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards: https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

transferFrom

```
1function transferFrom(
2        address from,
3        address to,
4        uint256 amount
5    ) external returns (bool);
```

Moves `amount` tokens from `from` to `to` using the allowance mechanism. `amount` is then deducted from the caller's allowance. Returns a boolean value indicating whether the operation succeeded.

IERC20Custom Interface

The IERC20Custom interface inherits from IERC20 and IERC20Metadata from v4.7.3 of OpenZeppelin and declares the following additional functions:

increaseAllowance

```
1function increaseAllowance(address agent, uint256 increasedAllowance) external return
s (bool);
```

Atomically increase the allowance for an agent by the caller (holder). This can mitigate the race condition noted above for {approve}. The agent’s address can’t be zero. The allowance can't be increased beyond max uint256.

decreaseAllowance

```
1function decreaseAllowance(address agent, uint256 decreasedAllowance) external return
s (bool);
```

Atomically decrease the allowance for an agent by the caller (holder). This can mitigate the race condition noted above for {approve}. The agent’s address can’t be zero. The allowance can't be decreased beyond zero.

IRenounceable Interface

renounceRole

```
1function renounceRole(bytes32 role) external;
```

Revokes the given role from own (calling) account.

IKYC Interface

grantKYC

```
1function grantKYC(address holder) external;
```

Called by KYCER_ROLE addresses to mark a holder address as KYCed.

revokeKYC

```
1function revokeKYC(address holder) external;
```

Called by KYCER_ROLE addresses to remove their KYC mark from the KYCed holder address.

ICheckKYC Interface

isKYCed

```
1function isKYCed(address holder) external view returns (bool);
```

Returns whether the holder’s address is KYCed (true) or not (false).

numKYCs

```
1function numKYCs(address holder) external view returns (uint256);
```

Returns number of times someone was KYCed

IHTLC Interface

createHTLC

```
1function createHTLC(
2        address receiver,
3        bytes32 hashlock,
4        uint256 timelock,
5        uint256 amount
6    ) external returns (address);
```

Creates a new HTLC.

createHTLCFor

```
1    function createHTLCFor(
2        address sender,
3        address receiver,
4        bytes32 hashlock,
5        uint256 timelock,
6        uint256 amount
7    ) external returns (address);
```

Allow an agent to create a HTLC on behalf of another account using available allowance.

withdrawHTLC

```
1function withdrawHTLC(HashTimeLockContract htlc, string memory preimage) external;
```

The receiver withdraws funds from HTLC.

refundHTLC

```
1 function refundHTLC(HashTimeLockCfunction htlcPreimage(HashTimeLockContract htlc) ex
ternal view returns (string memory);
```

Sender claims a refund from HTLC.

htlcPreimage

```
1function htlcPreimage(HashTimeLockContract htlc) external view returns (string memor
y);
```

Get preimage of HTLC.

htclSender

```
1function htlcSender(HashTimeLockContract htlc) external view returns (address);
```

Get sender of HTLC.

htclReciever

```
1function htlcReceiver(HashTimeLockContract htlc) external view returns (address);
```

get receiver of HTLC.

htclAmount

```
1function htlcAmount(HashTimeLockContract htlc) external view returns (uint256);
```

Get amount of HTLC.

htclBalance

```
1function htlcBalance(address htlcAddress) external view returns (uint256);
```

Get balance of HTLC.

htclHashLock

```
1function htlcHashLock(HashTimeLockContract htlc) external view returns (bytes32);
```

get hashlock of HTL.

htlcTimeLock

```
1function htlcTimeLock(HashTimeLockContract htlc) external view returns (uint256);
```

Get timelock of HTLC,

htlcWithdrawn

```
1 function htlcWithdrawn(HashTimeLockContract htlc) external view returns (bool);
```

Returns true if HTLC has been withdrawn.

htlcRefunded

```
1function htlcRefunded(HashTimeLockContract htlc) external view returns (bool);
```

Returns true if HTLC has been refunded.

htlcSeized

```
1function htlcSeized(address htlcAddress) external view returns (bool);
```

Returns true if HTLC funds have been seized.

getAllHTLCs

```
1function getAllHTLCs() external view returns (address[] memory);
```

Returns all HTLCs associated with msg.sender.

getActiveHTLCs

```
1function getActiveHTLCs() external view returns (address[] memory);
```

Returns all enabled and funded HTLCs associated with msg.sender.

getInActiveHTLCs

```
1 function getInactiveHTLCs() external view returns (address[] memory);
```

Returns all inactive (withdrawn, refunded, disabled or seized) HTLCs associated with msg.sender.

Hashed Timelock Contract (HTLC) Explained

The CBDC blockchain supports a HTLC mechanism to atomically swap assets in two separate systems (e.g. separate blockchains) for settlement. The HTLC capability of the CBDC system supports only the CBDC leg of an atomic settlement and does not directly control the other leg of a settlement. The HTLC mechanism works by using a time-based escrow that can be unlocked by providing a secret password that had been cryptographically hashed.

Trading assets across blockchains via HTLCs

There are five steps to trading assets across blockchains using HTLCs:

1. Party A chooses a secret password and encrypts it into a hashlock.
2. Party A creates a HTLC on their blockchain using the hashlock and deposits their funds into it.

3. Party B creates a separate HTLC on the CBDC system using the same hashlock and deposits their funds into it. (Party C may initiate this if they have an allowance over Party B's holdings).
4. Party A withdraws from the HTLC on the CBDC system by revealing the secret password.
5. Party B withdraws from the HTLC on Party A's blockchain by using the revealed password.

In the following scenario, Alice and Bob want to swap assets across different blockchains. Alice is swapping tokens in another blockchain with Bob for CBDC Tokens. Alice and Bob have accounts on the CBDC and also on the other blockchain. Alice has agreed with Bob to swap 5 other Tokens for 5 CBDC Tokens. This swap can be conducted atomically while reducing counterparty risk through HTLC contracts.

Step 1: Create the hashlock

- Alice creates a hashlock by choosing a secret password and encrypting it using the Keccak-256 hashing algorithm.
    - Keccak-256 Hash of 'SECRET' is b0ce8d7bebe80950cff1da1a8bed802684fb3e87c169b9f3c0af8801377c7a4c

Note that each password has to be unique in the CBDC blockchain. This means if someone has already used the password, it cannot be used again.

Step 2: Alice Creates the first HTLC on the other blockchain

- Alice creates the HTLC on the other blockchain system using the hashlock. The method to create HTLC on the foreign blockchain will be completely dependent on the unique chain/smart contract. However, you can expect to find a similar set of standard functions available (to create, withdraw, refund and verify the details of a HTLC).
- Alice needs to send the contract address to Bob so that he can verify the details match what they agreed on.
- Alice needs to provide Bob with the hashlock she created

Note that the hashing algorithm used in the other blockchain **MUST be keccak-256**. **The password will NOT unlock the hashlock if the hashing algorithms are**

different.

Step 3: Bob Creates the HTLC on the CBDC blockchain

- Bob needs to verify the details of the HTLC created by Alice on the other blockchain.
- Once Bob has verified the details of the HTLC match the agreed-upon details, Bob can create a HTLC on the CBDC blockchain.
- Bob creates a HTLC by calling the **createHTLC** function. This will lock away his funds. The arguments required by this function are:
    - The receiver address: Alices's address on the CBDC blockchain
    - The hashlock that was provided by Alice.
    - The timelock: an integer representing the UNIX timestamp that signifies when the contract expires
        - The UNIX timestamp for 1st January 2023 at 1.00 pm is 1672520400
    - The amount is the number of tokens that are to be locked away. Note that this number has to be less than or equal to Bob's balance or this transaction will fail

The expiry timelock on Bob's HTLC needs to be earlier than Alice's HTLC, so that Bob has enough time to withdraw from Alice's HTLC. Otherwise, if Bob waits until the last second to withdraw funds from Alice's HTLC (thereby revealing her secret password), Bob may not have enough time to withdraw from Alice's HTLC using the revealed password.

```
1function createHTLC(
2        address receiver,
3        bytes32 hashlock,
4        uint256 timelock,
5        uint256 amount
6    ) external returns (address);
```

- After successfully creating the HTLC. Bob must provide the HTLC address to Alice so that she can verify the details.
- Alice should query information about the HTLC created on the CBDC blockchain to verify the details match the agreed-upon swap by using the following

functions:

- **htlcSender**: check that this is Bob’s address
- **htlcReceiver**: check that this is her address
- **htlcAmount**: check that the amount Bob deposited is the amount they agreed on
- **htlcBalance**: check that the HTLC is still funded
- **htlcHashLock**: check the hashlock is what she provided
- **htlcTimelock**: check that the timelock is what they agreed on
- **htlcWithdrawn**: check that the HTLC hasn’t been withdrawn
- **htlcRefunded**: check that the HTLC hasn’t been refunded
- **htlcSeized**: check that the HTLC hasn’t been seized

At this point, Bob can’t withdraw Alice’s funds because he doesn’t know her secret password.

Step 4: Alice withdraws from the HTLC on the CBDC system

- Once Alice has verified the details of Bob’s HTLC, Alice withdraws the funds from Bob’s HTLC using the password of the hashlock using the **withdrawHTLC** function.
- This will transfer the tokens from the HTLC to Alice’s wallet in the CBDC blockchain. Once she withdraws the funds, the secret phrase will be revealed on the CBDC blockchain (see the error panel below). Bob will then be able to use the password to unlock Alice’s HTLC and retrieve the funds.

If Alice doesn't withdraw funds from Bob’s HTLC before it expires, Bob can reclaim his funds with a refund.

Step 5: Bob withdraws from the HTLC on the other blockchain

- Bob looks up the revealed password saved in his HTLC and uses it to withdraw from Alice’s HTLC in the other blockchain using the **withdrawHTLC** function.
- If Bob doesn't withdraw his funds before Alice’s HTLC expires, she can withdraw her funds using **refundHTLC**.

Whoever chooses the secret password will create the first HTLC and have funds locked away the longest, but will also receive the other party’s funds first. It is possible for Bob to choose a secret password instead of Alice.