OpenCoin

Release 0.4

The OpenCoin Crew

22.07.2022, 16:18

CONTENTS:

A protocol for privacy preserving electronic cash payments

Version: 0.4 - draft (July 2022)

Warning: This document really is a draft - it is still changing, right as you read this.

We will remove this warning when it is not longer necessary.

For more information go to **https://opencoin.org**, or mail us at collective@opencoin.org

You can read the documentation as a website or as as pdf.

CHAPTER ONE

OVERVIEW

We propose a protocol that allows cash-like payments in the electronic world. It is based on the invention by David Chaum[1]. This means that we define message structures around blind signatures, that focus on *untraceable* payments. 'Untracable' because even though there is a central entity (called the issuer, something like a bank), this central entity can't see the transactions happening.

The focus of the project is the protocol. This means we standardize the way we exchange messages and their content in order to make electronic cash payments. But we don't deliver an implementation here. That is the scope of other project(s). OpenCoin is the foundation to build upon.

1.1 What can I do with it?

The most common use case is "electronic cash", that is an electronic currency that allows fast untraceable transactions and is convertible to a "real" currency. But other uses can be imagined as well: a time bank, carbon based emission tokens, or electronic voting, to name a few. Everybody can run an issuer and can set their own rules[2]

1.2 How does it work?

The simplified overview show a high level description of the basic system. We have three participants: Alice and Bob are normal users, while the Issuer is something like a bank, capable of minting coins. It also acts as an exchange for 'real-world' currency. At this high level it works as follows:

1. Alice asks the Issuer to **mint** coins. This is done in a special way using *blind signatures*, which means that the coins *can't be linked to her* later on.
2. Alice then **transfers** the coins to Bob. She can do the transfer any way she wants, e.g. using a messenger like Signal, Email or any other system of her choice (also depending

[1] David Chaum, "Blind signatures for untraceable payments", Advances in Cryptology - Crypto '82, Springer-Verlag (1983), 199-203.

[2] Please check with your lawyer if this is a good idea.

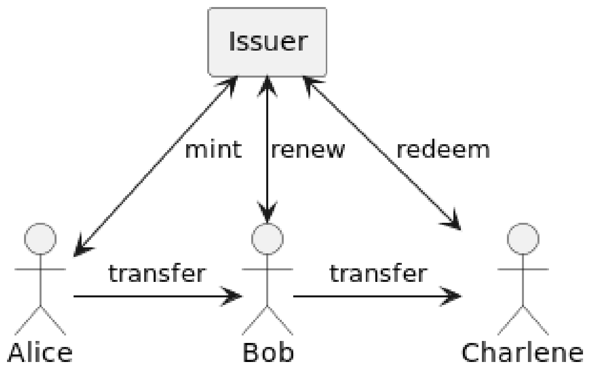


Fig. 1: Simplified overview of the OpenCoin flow.

on what her client software supports). This could even be done by printing the coins and handing them over.

3. Bob then **renews** the coins. He swaps the coins he got from Alice for fresh coins at the Issuer. This way he protects himself against Alice "accidentally" using the coins somewhere else. The Issuer marks used coins in its database. This way one can spend opencoins only once, and *double spending* is ruled out.
4. Bob might **transfer** the coins to yet another person, Charlene.
5. Charlene decides to **redeem** the coins, meaning she asks the issuer to convert the opencoins for real-world money.

On **blind signatures**: at the core of a coin is a serial number with a signature from the mint. In order to ensure that *a coin can't be traced back to the original client* we use blind signatures:

Imagine Alice wants to give Bob a coin that the issuer can't trace. In order to ensure the coin can't be traced back to Alice, the issuer needs to sign the serial number without seeing it. In non-technical terms Alice puts the serial number in an envelope (along with carbon copy paper), and the issuer actually signs the envelope. Because of the carbon copy paper the signature presses through onto the serial number. Alice can then open up the envelope and has a signed serial, without the issuer ever seeing it.

1.3 Who is it for?

OpenCoin (the protocol) allows the development of applications for electronic cash. So firstly OpenCoin is targeted at developers. These applications in return should allow everyone to make and receive electronic payments. It still requires somebody to run the central issuer. This issuer would run an OpenCoin based electronic money system. Because electronic money is quite regulated in Europe (and other countries), the issuer would be most likely a regulated electronic money provider or a bank. We think, that a central bank would be the best issuer, because central banks issue money anyhow. But nothing technical stops you from using OpenCoin for your private project[3].

[3] Again, please check with your lawyer if this is a good idea.

1.4 Alternatives

Why don't just use one of the alternatives?

1.4.1 Bitcoin / blockchain

Bitcoin (or blockchain in the more general form) is basically the opposite of OpenCoin: transfers have to happen within the system, they are visible to everybody, there is no central instance, there is no guaranteed value you can redeem the bitcoins for.

OpenCoin on the contrary makes the transfers invisible and untraceable, and has a central instance that is able to guarantee a value if you redeem the OpenCoin.

Also, Bitcoin is environmentally insane.

1.4.2 GNU Taler

GNU Taler is build around the same central idea as OpenCoin. It started later, and is more complete than OpenCoin. GNU Taler differs in the way the system takes care of the *renewal step* and it introduces coin splitting. It also makes more assumptions regarding the clients, the clients have defined roles (e.g. consumer and merchant). GNU Taler is also much more a finished product than OpenCoin.

OpenCoin is a protocol, a shared ground to build multiple implementations upon.

We also think that some issues (like tax-ability) could be solved to some extent within OpenCoin, but are *better solved outside the transfer system anyhow*.

CHAPTER TWO

OPENCOIN PROTOCOL

2.1 Assumptions

The exchange of messages MUST happen over a **secure channel**. For HTTPS this means TLS, but other channels, like messengers, will most likely provide their own. For an email exchange GPG would be recommended.

Either way, it is the responsibility of the developer to take care of the transport security.

When requesting the issuer to mint or redeem coins some form of **authentication & authorization** is most likely required - the issuer needs to secure payment for the coins, or make a payment somewhere for redeemed coins. Because auth* might already be provided by the transport layer, we don't include it in the OpenCoin protocol.

2.2 Sequence diagram

The following *OpenCoin sequence diagram* shows the flow of messages (and actions).

2.3 Protocol description

This is a description of the actual steps. Additional info for individual steps can also be found in the respective sections in *Schemata*.[1]

[1] We find it easier to follow along with the above diagram open in a second window (or printout).

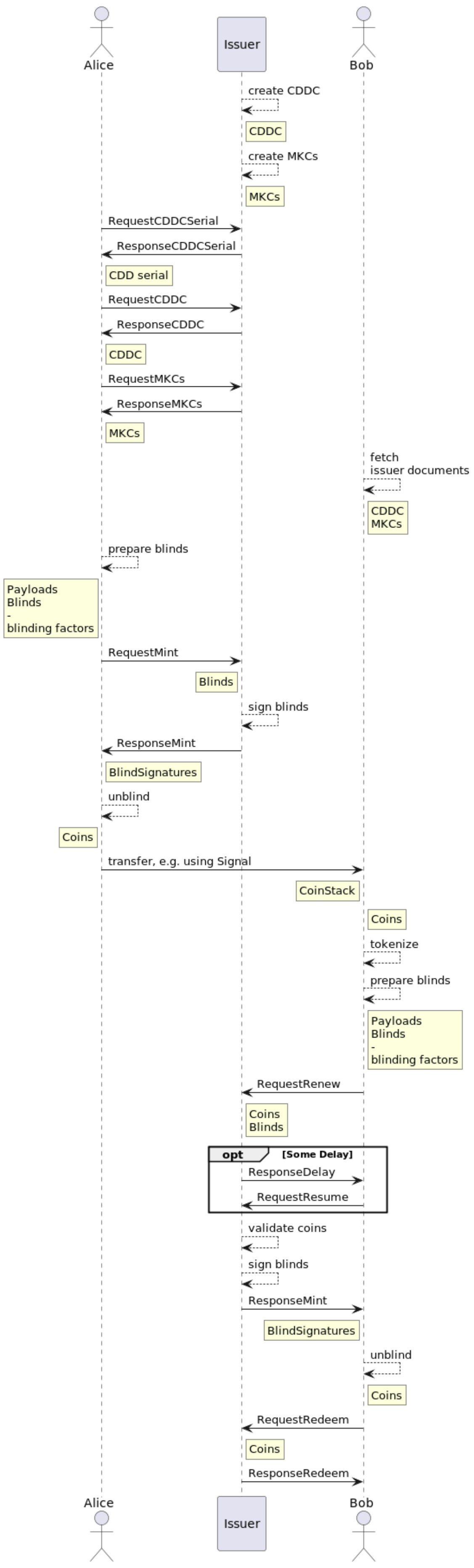


Fig. 1: OpenCoin sequence diagram

2.3.1 Participants

Issuer can mint, refresh and redeem coins. This entity will probably have an account handling system (a.k.a. bank) behind it for doing actual real-world payments. The issuer is trusted to handle coins and payments correctly, but is *not trusted* regarding privacy - the target of OpenCoin is to protect the privacy of the transfers.

Alice and **Bob** are clients using OpenCoin. Technically they are client software called "wallets", and they represent the *users* of the system.[2] They need to be authenticated by the Issuer in order to mint or redeem coins. To renew coin authentication is optional, and would allow a "closed" system, in which accounts of the users could be monitored quite closely.

2.3.2 Steps in the protocol

create CDDC

The issuer creates a pair of cryptographic keys (the currency keys), and signs a *Currency Description Document Certificate* (*CDDC*) with its secret key. The CDDC contains information about the currency, like denominations, urls but also the public key. This is the top document which establishes the trust in all other elements.

Not mentioned in the CDDC but probably somewhere on the issuer website are the rules for the relation between opencoins and actual real-world money. Let's say the currency of an example issuer is called "opencent".[3] The rule might be that one opencent is given out for 0.01 EUR , and redeemed for 0.01 EUR, effectively binding the opencent to the EUR.

create MKCs

For each denomination in the currency separate minting keys are generated, and a *Mint Key Certificate* (*MKC*) for them as well. Those MKCs are signed using the secret currency key. The mint keys are only valid for a defined period of time.[4]

RequestCDDCSerial

RequestCDDCSerial asks for the current serial number of the CDDC. The currency description could change over time, maybe because urls have changed. On every change a new CDDC is created, with a new, increasing serial number. The clients need to make sure to always use the most current CDDC, but they can cache it, allowing them to skip the next step.

[2] To keep the diagram simple we have left out Charlene who was mentioned above in "*How does it work?*". Bob does everything she does.

[3] "opencent" refers to the specific example currency. The generic term "opencoin" refers to any currency following the OpenCoin protocol (of which opencent is one).

[4] This is to minimize damage in case the mint keys get compromised.

ResponseCDDCSerial

ResponseCDDSerial contains the current serial of the *CDDC*.

RequestCDDC

RequestCDDC asks for a *CDDC*. If no serial is provided, the message asks for the most current CDDC.

ResponseCDDC

ResponseCDDC contains the *CDDC*

RequestMKCs

RequestMKCs asks for the *Mint Key Certificates*. The client can specify specific denominations or *mint key ids*. An unspecified request will return all current MKCs.

ResponseMKCs

ResponseMCKs contains the *MKCs*

prepare blinds

This step prepares a coin. In essence this is a *Payload* containing a serial number, which is later on signed by the issuer using a denomination specific mint key. The "envelope" *mentioned above* really means that the serial is blinded using a separate random secret **blinding factor** for each serial number. This factor is needed later on to "open up the envelope", reversing the blinding operation. Hence, the client has to store the blinding factor for later on. As the blinding factor is individual for each serial number, a reference number is created to reference serial, blinding factor and *Blind*.

The blinds contain the reference, the Blind to be signed, and the mint key id for the denomination or value of the coin.

RequestMint

RequestMint hands in the *Blinds* created in the step before, asking for the blind to be signed.

Most likely the issuer has authenticated the client. The mint key id tells the Issuer what denomination to use for the signing operation. This will allow the Issuer to deduct a payment for the minting operation (outside OpenCoin).

The message also carries a transaction_reference (a random number), which is used in case of a delay in the minting process. The client can then later on ask again for the signatures to be delivered using the same transaction_reference.

sign blinds

The issuer uses the secret minting key for the desired operation to sign the *Blind*, creating *Blind Signatures*.

ResponseMint

ResponseMint contains the *Blind Signatures* for the *Blinds*.

unblind

The client will unblind the *Blind Signature* using the before stored secret blinding factor. This gives the client the signature for the serial number, and both together make up the *Coin*. The signature is validated by Alice.

CoinStack

When sending coins, multiple coins can be combined into a *CoinStack*. This CoinStack can also have a “subject”, a note on the reason the CoinStack is handed over in the first place, maybe containing an order reference.

The transfer of the CoinStack is out of scope of the OpenCoin protocol. We imagine multiple ways: using a messenger like Signal, using email or using the Browser. A CoinStack can also be encoded using a QR code, and maybe printed out and sent using normal postal mail.

Anyhow, the point of this step is that Alice transfers a CoinStack to Bob. And because she is a fair user, she will delete all coins that were contained in the CoinStack on her side.

tokenize

Coins that are received need to be swapped for new ones, in order to protect the receiver against double spending. Bob needs to decide which new coin sizes he needs to have. He tokenizes the amount in the right way to have a good selection of future coin sizes (he wants the right change in his wallet to pay all possible future amounts).[5]

[5] It might be that also some existing coins might be needed to be swapped to get a good coin selection. See *RequestRenew Message*.

prepare blinds

Knowing the right coin selection from the step before Bob prepares *Blinds* the same way Alice has done with hers, creating *Payloads* (containing serials), and indexing the blinding secrets using a reference.

RequestRenew

The renewal process is effectively the same as in minting new coins, but it is paid for in opencoins, instead of making a payment in the background using accounts. Hence, the *RequestRenew* message needs to contain *Coins* that have a value that matches the sum of value of the *Blinds*. The message also contains a transaction_reference in case a delay happens.

ResponseDelay

This step is optional.

If something takes a while at the issuer when signing the blinds, either while handling a *RequestMint* or *RequestRenew* a *ResponseDelay* can be sent back to indicate that the client should try again sometime later. This allows the network connection be closed in the meantime. Hopefully operations resume in a short time.

Delays should be avoided on the issuer side.

RequestResume

Bob will try some time later on to resume the transaction (the renewal in this case). The *RequestResume* message will send over the transaction reference, and the issuer will hopefully respond with a *ResponseMint* message, or with another *ResponseDelay*.

validate coins

Bob validates the *Coins*, just as Alice did.

RequestRedeem

Bob might want to convert some or all of the *Coins* he holds for real-world currency at the issuer. He sends in the coins in a *RequestRedeem* message. This effectively takes the coins out of circulation, and the issuer will make a payment to Bob's account. This requires the client to be authenticated for this step, which again is outside the OpenCoin protocol.

ResponseRedeem

The issuer confirms that everything went ok using the *ResponseRedeem* message.

CHAPTER THREE

SCHEMATA

In here we present the actual schemata for the all the messages that we send and receive in the OpenCoin protocol. Each schema contains a description, a list of its fields and an example. The field names are linked to the *fields page*, where you can more detailed information about the fields.

Note: The examples are produced by the run_protocol.py script.

3.1 Elements of messages

3.1.1 CDD

The Currency Description Document holds all information about a currency that is following the OpenCoin protocol.

It contains the master key, the denominations, the services, the name of the currency and more.

Always part of a *CDDC*

Fields

- ***additional_info***: A field where the issuer can store additional information about the currency. Free text for humans. *(String)*
- ***cdd_expiry_date***: The date the CDD expires. *(String)*
- ***cdd_location***: Location to download the CDD from. *(URL)*
- ***cdd_serial***: The version of the CDD. *(Int)*
- ***cdd_signing_date***: When was the CDD signed? *(DateTime)*
- ***currency_divisor***: Used to express the value in units of 'currency name'. *(Int)*
- ***currency_name***: The name of the currency, e.g. Dollar. *(String)*
- ***denominations***: The list of possible denominations. *(List of Int)*

- *id*: Identifier, a somewhat redundant hash of the *PublicKey* *(BigInt)*
- *info_service*: A list of locations where more information about the currency can be found. *(WeightedURLList)*
- *redeem_service*: A list of locations where *Coins* can be redeemed. *(WeightedURLList)*
- *issuer_cipher_suite*: Identifier of the cipher suite that is used. *(String)*
- *issuer_public_master_key*: The hash of the issuer's public key *(*PublicKey*)*
- *protocol_version*: The protocol version that was used. *(Url)*
- *renew_service*: A list of locations where *Coins* can be renewed. *(WeightedURLList)*
- *type*: String identifying the type of message. *(String)*
- *mint_service*: A list of locations where *Blinds* can be minted into *Coins* *(WeightedURL-List)*

Example

```
{
  "additional_info": "",
  "cdd_expiry_date": "2023-07-22T15:45:53.164685",
  "cdd_location": "https://opencent.org",
  "cdd_serial": 1,
  "cdd_signing_date": "2022-07-22T15:45:53.164685",
  "currency_divisor": 100,
  "currency_name": "OpenCent",
  "denominations": [1, 2, 5],
  "id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
  "info_service": [
        [10, "https://opencent.org"]
    ],
  "issuer_cipher_suite": "RSA-SHA256-PSS-CHAUM82",
  "issuer_public_master_key": {
    "modulus": "daaa63ddda38c189b8c49020c8276adbe0a695685a...",
    "public_exponent": 65537,
    "type": "rsa public key"
  },
  "mint_service": [
        [10, "https://opencent.org"],
    [20, "https://opencent.com/validate"]
    ],
  "protocol_version": "https://opencoin.org/1.0",
  "redeem_service": [
        [10, "https://opencent.org"]
    ],
  "renew_service": [
```

(continues on next page)

(continued from previous page)

```
        [10, "https://opencent.org"]
      ],
  "type": "cdd"
}
```

Complete example: cdd.json

Secret key

To make this documentation completely reproducible, here is the data for the secret issuer key:

```
{
  "d": "19ed9857e71751c39dd818eafa30ca57f6246a94ec...",
  "n": "daaa63ddda38c189b8c49020c8276adbe0a695685a..."
}
```

Complete example: issuer_secret.json

Warning: Don't ever publish your secret keys!

3.1.2 CDDC

The certificate for the *CDD*, signed with the secret master key. This is the "trust anchor".

Fields

- ***cdd***: Contains the Currency Description Document (CDD). *(CDD)*
- ***signature***: A signature within a certificate. *(String)*
- ***type***: String identifying the type of message. *(String)*

Example

```
{
  "cdd": {
    "additional_info": "",
    "cdd_expiry_date": "2023-07-22T15:45:53.164685",
    "cdd_location": "https://opencent.org",
    "cdd_serial": 1,
    "cdd_signing_date": "2022-07-22T15:45:53.164685",
    "currency_divisor": 100,
```

(continues on next page)

(continued from previous page)

```
    "currency_name": "OpenCent",
    "denominations": [1, 2, 5],
    "id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
    "info_service": [
        [10, "https://opencent.org"]
      ],
    "issuer_cipher_suite": "RSA-SHA256-PSS-CHAUM82",
    "issuer_public_master_key": {
      "modulus": "daaa63ddda38c189b8c49020c8276adbe0a695685a...",
      "public_exponent": 65537,
      "type": "rsa public key"
    },
    "mint_service": [
        [10, "https://opencent.org"],
      [20, "https://opencent.com/validate"]
      ],
    "protocol_version": "https://opencoin.org/1.0",
    "redeem_service": [
        [10, "https://opencent.org"]
      ],
    "renew_service": [
        [10, "https://opencent.org"]
      ],
    "type": "cdd"
  },
  "signature": "3fc265677973b6308dd665b66750cd27911e8a7b5c...",
  "type": "cdd certificate"
}
```

Complete example: cddc.json

3.1.3 PublicKey

Schema to hold public keys. We have decided for our own format because we want to keep things simple. Using a predefined format would probably lead to use bigger libraries when implementing the protocol.

Always part of a *CDD* or *MKC*.

Fields

- *modulus*: The modulus of the public key *(BigInt)*
- *public_exponent*: The exponent of the public key. *(BigInt)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "modulus": "daaa63ddda38c189b8c49020c8276adbe0a695685a...",
  "public_exponent": 65537,
  "type": "rsa public key"
}
```

Complete example: issuer_public_master_key.json

3.1.4 MintKey

This describes a key that is used to sign/mint coins for a single denomination. As the key doesn't see the content it is signing (it is blinded, after all), we need to define the meaning of its signature: "this signature is worth *n* units of value".

The key will be used for a period of time, after which it is going to be swapped for a new one. The coins will be valid a bit longer then the signing time of the key.

Warning: **Never** use the mint key for encryption, **only** ever for signing!

Always part of an *MKC*

Fields

- *cdd_serial*: The version of the CDD. *(Int)*
- *coins_expiry_date*: Coins expire after this date. *(DateTime)*
- *denomination*: The value of the coin(s). *(Int)*
- *id*: Identifier, a somewhat redundant hash of the *PublicKey* *(BigInt)*
- *issuer_id*: The identifier (hash) of the issuer public master key in the CDDC *(BigInt)*
- *public_mint_key*: The public key of the mint key. *(PublicKey)*
- *sign_coins_not_after*: Use *MintKey* only before this date. *(DateTime)*
- *sign_coins_not_before*: Use *MintKey* only after this date. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "cdd_serial": 1,
  "coins_expiry_date": "2023-10-30T15:45:53.164685",
  "denomination": 1,
  "id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
  "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
  "public_mint_key": {
    "modulus": "ce3af9b91d6a6da0fe8def8870743428c0c4c60f5a...",
    "public_exponent": 65537,
    "type": "rsa public key"
  },
  "sign_coins_not_after": "2023-07-22T15:45:53.164685",
  "sign_coins_not_before": "2022-07-22T15:45:53.164685",
  "type": "mint key"
}
```

Complete example: mintkey_1.json

Secret key

To make this documentation completely reproducible, here is the data for the secret mint key for the value 1:

```
{
  "d": "31d379e649bf1f0198befe327ec8f4992c09deb872...",
  "n": "ce3af9b91d6a6da0fe8def8870743428c0c4c60f5a..."
}
```

Complete example: mintkey_1_secret.json

Warning: Don’t ever publish your secret keys!

3.1.5 MKC

A *Mint Key Certificate* for the *MKC*. Signed with the secret master key in the *CDD*.

Fields

- *mint_key*: The mint key that was signed in the certificate. *(MintKey)*
- *signature*: A signature within a certificate. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "mint_key": {
    "cdd_serial": 1,
    "coins_expiry_date": "2023-10-30T15:45:53.164685",
    "denomination": 1,
    "id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
    "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
    "public_mint_key": {
      "modulus": "ce3af9b91d6a6da0fe8def8870743428c0c4c60f5a...",
      "public_exponent": 65537,
      "type": "rsa public key"
    },
    "sign_coins_not_after": "2023-07-22T15:45:53.164685",
    "sign_coins_not_before": "2022-07-22T15:45:53.164685",
    "type": "mint key"
  },
  "signature": "84aceee2562297d127560bff4d66654e881e89eae2...",
  "type": "mint key certificate"
}
```

Complete example: mkc_1.json

3.1.6 Payload

These are the "innards" of *Coin*. The mint key id is only a helper so that the corresponding key for the signature of the coin can be found faster. The same is true for the denomination - an implementation should take the coins value from the *MintKey*, and not from the contents of the Payload, because the values in the Payload are user defined.

Fields

- ***cdd_location***: Location to download the CDD from. *(URL)*
- ***denomination***: The value of the coin(s). *(Int)*
- ***issuer_id***: The identifier (hash) of the issuer public master key in the CDDC *(BigInt)*
- ***mint_key_id***: Identifier of the mint key used. *(BigInt)*
- ***protocol_version***: The protocol version that was used. *(Url)*
- ***serial***: The serial of the *Coin*. *(BigInt)*
- ***type***: String identifying the type of message. *(String)*

Example

```
{
  "cdd_location": "https://opencent.org",
  "denomination": 1,
  "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
  "mint_key_id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
  "protocol_version": "https://opencoin.org/1.0",
  "serial": "cd613e30d8f16adf91b7584a2265b1f5",
  "type": "payload"
}
```

Complete example: payload_a0.json

3.1.7 Blind

Contains the blinded hash for a *Payload*, and says which mint key to use. The reference is needed to connect signatures to blinds later on (we don't want to rely on list orders).

Fields

- *blinded_payload_hash*: The blinded hash of a payload. *(BigInt)*
- *mint_key_id*: Identifier of the mint key used. *(BigInt)*
- *reference*: An identifier that connects *Blind*, *BlindSignature* and blinding secrets. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "blinded_payload_hash": "924edb672c3345492f38341ff86b57181da4c673ef..
↪.",
  "mint_key_id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
  "reference": "a0",
  "type": "blinded payload hash"
}
```

Complete example: blind_a0.json

3.1.8 BlindSignature

The signature for a blind. The blind is specified with the reference.

Fields

- *blind_signature*: The signature on the blinded hash of a payload. *(BigInt)*
- *reference*: An identifier that connects *Blind*, *BlindSignature* and blinding secrets. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "blind_signature": "57744b5430075756d246ceff94f47e9ffeb80af3d8...",
  "reference": "a0",
  "type": "blind signature"
}
```

Complete example: blind_signature_a0.json

3.1.9 Coin

The certificate for a payload. The signature is the unblinded *BlindSignature*.

Fields

- ***payload***: The payload of the coin. *(Payload)*
- ***signature***: A signature within a certificate. *(String)*
- ***type***: String identifying the type of message. *(String)*

Example

```
{
  "payload": {
    "cdd_location": "https://opencent.org",
    "denomination": 1,
    "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
    "mint_key_id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
    "protocol_version": "https://opencoin.org/1.0",
    "serial": "cd613e30d8f16adf91b7584a2265b1f5",
    "type": "payload"
  },
  "signature": "2ec0af339566b19fb9867b491ce58025dcefcab649...",
  "type": "coin"
}
```

Complete example: coin_a0.json

3.2 Messages

3.2.1 RequestCDDSerial Message

This message asks for the cdd_serial of the current *CDDC*.

Fields

- ***message_reference***: Client internal message reference. *(Integer)*
- ***type***: String identifying the type of message. *(String)*

Example

```
{
  "message_reference": 100000,
  "type": "request cdd serial"
}
```

Complete example: request_cddc_serial.json

3.2.2 ResponseCDDSerial Message

Returns the current cdd_serial.

Fields

- *cdd_serial*: The version of the CDD. *(Int)*
- *message_reference*: Client internal message reference. *(Integer)*
- *status_code*: The issuer can return a status code, like in HTTP: *(Integer)*
- *status_description*: Description that the issuer passes along with the status_code. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "cdd_serial": 1,
  "message_reference": 100000,
  "status_code": 200,
  "status_description": "ok",
  "type": "response cdd serial"
}
```

Complete example: response_cddc_serial.json

3.2.3 RequestCDDC Message

This requests the *CDDC* specified by the cdd_serial. If cdd_serial is set to 0, the most current CDDC is returned.

Fields

- *cdd_serial*: The version of the CDD. *(Int)*
- *message_reference*: Client internal message reference. *(Integer)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "cdd_serial": 1,
  "message_reference": 100001,
  "type": "request cddc"
}
```

Complete example: request_cddc.json

3.2.4 ResponseCDDC Message

This response carries the *CDDC*.

Fields

- *cddc*: A full Currency Description Document Certificate. *(CDDC)*
- *message_reference*: Client internal message reference. *(Integer)*
- *status_code*: The issuer can return a status code, like in HTTP: *(Integer)*
- *status_description*: Description that the issuer passes along with the status_code. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "cddc": {
    "cdd": {
      "additional_info": "",
      "cdd_expiry_date": "2023-07-22T15:45:53.164685",
      "cdd_location": "https://opencent.org",
      "cdd_serial": 1,
      "cdd_signing_date": "2022-07-22T15:45:53.164685",
      "currency_divisor": 100,
      "currency_name": "OpenCent",
      "denominations": [1, 2, 5],
```

(continues on next page)

(continued from previous page)

```
      "id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
      "info_service": [
        [10, "https://opencent.org"]
      ],
      "issuer_cipher_suite": "RSA-SHA256-PSS-CHAUM82",
      "issuer_public_master_key": {
        "modulus": "daaa63ddda38c189b8c49020c8276adbe0a695685a...",
        "public_exponent": 65537,
        "type": "rsa public key"
      },
      "mint_service": [
        [10, "https://opencent.org"],
        [20, "https://opencent.com/validate"]
      ],
      "protocol_version": "https://opencoin.org/1.0",
      "redeem_service": [
        [10, "https://opencent.org"]
      ],
      "renew_service": [
        [10, "https://opencent.org"]
      ],
      "type": "cdd"
    },
    "signature": "3fc265677973b6308dd665b66750cd27911e8a7b5c...",
    "type": "cdd certificate"
  },
  "message_reference": 100001,
  "status_code": 200,
  "status_description": "ok",
  "type": "response cddc"
}
```

Complete example: response_cddc.json

3.2.5 RequestMKCs Message

This requests one or more *MKCs*. The fields *denominations* and *mint_key_ids* specify which MKCs should be delivered. Denominations refer to the most current key(s) for the given denominations. If both fields are empty, all most current MKCs are delivered - we assume that is the normal use case.

Fields

- *denominations*: The list of possible denominations. *(List of Int)*
- *message_reference*: Client internal message reference. *(Integer)*
- *mint_key_ids*: What mint keys should be returned? *(List of BigInt)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "denominations": [1, 2, 5],
  "message_reference": 100002,
  "mint_key_ids": [],
  "type": "request mint key certificates"
}
```

Complete example: request_mkc.json

3.2.6 ResponseMKCs Message

This delivers the [MKCs] as specified in the *RequestMKCs*.

Fields

- *keys*: A list of Mint Key Certificates *(List of* MKCs*)*
- *message_reference*: Client internal message reference. *(Integer)*
- *status_code*: The issuer can return a status code, like in HTTP: *(Integer)*
- *status_description*: Description that the issuer passes along with the status_code. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "keys": [
    {
      "mint_key": {
        "cdd_serial": 1,
        "coins_expiry_date": "2023-10-30T15:45:53.164685",
        "denomination": 1,
        "id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
```

(continues on next page)

(continued from previous page)

```
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "public_mint_key": {
          "modulus": "ce3af9b91d6a6da0fe8def8870743428c0c4c60f5a...",
          "public_exponent": 65537,
          "type": "rsa public key"
        },
        "sign_coins_not_after": "2023-07-22T15:45:53.164685",
        "sign_coins_not_before": "2022-07-22T15:45:53.164685",
        "type": "mint key"
      },
      "signature": "84aceee2562297d127560bff4d66654e881e89eae2...",
      "type": "mint key certificate"
    },
    {
      "mint_key": {
        "cdd_serial": 1,
        "coins_expiry_date": "2023-10-30T15:45:53.164685",
        "denomination": 2,
        "id": "f2864e5cd937dbaa4825e73a81062de162143682",
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "public_mint_key": {
          "modulus": "ca03c4fa5a94144c5649d1f5d7cb9780e45911323c...",
          "public_exponent": 65537,
          "type": "rsa public key"
        },
        "sign_coins_not_after": "2023-07-22T15:45:53.164685",
        "sign_coins_not_before": "2022-07-22T15:45:53.164685",
        "type": "mint key"
      },
      "signature": "2db1d5a728a841f415f414a07767b7d43ab3d62235...",
      "type": "mint key certificate"
    },
    {
      "mint_key": {
        "cdd_serial": 1,
        "coins_expiry_date": "2023-10-30T15:45:53.164685",
        "denomination": 5,
        "id": "897a16bf12bd9ba474ef7be0e3a53553a7b4ece8",
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "public_mint_key": {
          "modulus": "d35f9a322c851eb071f557f8b5723d10c7889ad7ba...",
          "public_exponent": 65537,
          "type": "rsa public key"
        },
        "sign_coins_not_after": "2023-07-22T15:45:53.164685",
        "sign_coins_not_before": "2022-07-22T15:45:53.164685",
```

(continues on next page)

(continued from previous page)

```
        "type": "mint key"
      },
      "signature": "9a76c1f3223817de54632b9f2eb5c94a0426d26738...",
      "type": "mint key certificate"
    }
  ],
  "message_reference": 100002,
  "status_code": 200,
  "status_description": "ok",
  "type": "response mint key certificates"
}
```

Complete example: response_mkc.json

3.2.7 RequestMint Message

Request blinds to be signed. The *Blinds* hold the information which mint keys are to be used for minting.

The client asking for this action should be authenticated, and the issuer should check if the client meets the requirements of the request, a.k.a. has enough funds to pay for the minting process.

Fields

- *blinds*: A List of Blinds. *(List of* Blinds*)*
- *message_reference*: Client internal message reference. *(Integer)*
- *transaction_reference*: A random identifier that allows the client to resume a delayed mint/renew process. *(BigInt)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "blinds": [
    {
      "blinded_payload_hash":
↪"924edb672c3345492f38341ff86b57181da4c673ef...",
      "mint_key_id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
      "reference": "a0",
      "type": "blinded payload hash"
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "blinded_payload_hash":
↪"95db92e1c46ebea5edec5e508a831263de6fb78b4c...",
      "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
      "reference": "a1",
      "type": "blinded payload hash"
    },
    {
      "blinded_payload_hash":
↪"10afac98ac43eb40e996c621d5db4d2238348e3f74...",
      "mint_key_id": "897a16bf12bd9ba474ef7be0e3a53553a7b4ece8",
      "reference": "a2",
      "type": "blinded payload hash"
    }
  ],
  "message_reference": 100003,
  "transaction_reference": "b2221a58008a05a6c4647159c324c985",
  "type": "request mint"
}
```

Complete example: request_mint.json

3.2.8 ResponseMint Message

This delivers the *BlindSignatures*. The client will derive the signature for the *Coins* by unblinding the BlindSignatures.

Fields

- ***blind_signatures***: A list of BlindSignatures. *(List of* BlindSignatures*)*
- ***message_reference***: Client internal message reference. *(Integer)*
- ***status_code***: The issuer can return a status code, like in HTTP: *(Integer)*
- ***status_description***: Description that the issuer passes along with the status_code. *(String)*
- ***type***: String identifying the type of message. *(String)*

Example

```
{
  "blind_signatures": [
    {
      "blind_signature": "57744b5430075756d246ceff94f47e9ffeb80af3d8...
↪",
      "reference": "a0",
      "type": "blind signature"
    },
    {
      "blind_signature": "568bcaa9357d0fce7649c9217f275d277bb89721f4...
↪",
      "reference": "a1",
      "type": "blind signature"
    },
    {
      "blind_signature": "a8206a048987d9372232b7a084325ca26b4b151dda...
↪",
      "reference": "a2",
      "type": "blind signature"
    }
  ],
  "message_reference": 100003,
  "status_code": 200,
  "status_description": "ok",
  "type": "response mint"
}
```

Complete example: response_mint_a.json

3.2.9 CoinStack Message

This holds a set of coins. It is not a message in the strict sense, but more a transport container. Transferring the CoinStack in a manner that is untraceable by the issuer is key to protecting the privacy. How this is achieved is outside the protocol, and left as an exercise to the implementer :-)

Fields

- ***coins***: A list of coins. *(List of* Coins*)*
- ***subject***: A message that can be passed along with the coin stack. *(String)*
- ***type***: String identifying the type of message. *(String)*

Example

```
{
  "coins": [
    {
      "payload": {
        "cdd_location": "https://opencent.org",
        "denomination": 1,
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "mint_key_id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
        "protocol_version": "https://opencoin.org/1.0",
        "serial": "cd613e30d8f16adf91b7584a2265b1f5",
        "type": "payload"
      },
      "signature": "2ec0af339566b19fb9867b491ce58025dcefcab649...",
      "type": "coin"
    },
    {
      "payload": {
        "cdd_location": "https://opencent.org",
        "denomination": 2,
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
        "protocol_version": "https://opencoin.org/1.0",
        "serial": "78e510617311d8a3c2ce6f447ed4d57b",
        "type": "payload"
      },
      "signature": "6aefa7472518ed0a1ec64971220ce3a3a921a70bb0...",
      "type": "coin"
    },
    {
      "payload": {
        "cdd_location": "https://opencent.org",
        "denomination": 5,
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "mint_key_id": "897a16bf12bd9ba474ef7be0e3a53553a7b4ece8",
        "protocol_version": "https://opencoin.org/1.0",
        "serial": "e4b06ce60741c7a87ce42c8218072e8c",
        "type": "payload"
```

(continues on next page)

(continued from previous page)

```
      },
      "signature": "72da93670f666c529f26fcf15092a63c0fa48c8387...",
      "type": "coin"
    }
  ],
  "subject": "a little gift",
  "type": "coinstack"
}
```

Complete example: coinstack.json

3.2.10 RequestRenew Message

Coins that are received need to be swapped for new ones, in order to protect the receiver against double spending. Otherwise, the sender could keep a copy of the coins and try to use the coins again. Before doing so we need to ask: what coin sizes should be chosen for the coins to be minted?

What if we have not the right coin selection ("change") for an amount to pay? Imagine that the price is 5 opencent, but we just have coins in the sizes: 2, 2, 2.

One solution would be to require the recipient to give change back to the client. This would complicate the protocol, and would just shift the problem to the recipient. Another approach is to allow partial spending coins, but this again makes the protocol more complicated.[1]

The easy way out is to aim for a selection of coins that allows us to pay *any* amount below or equal to the sum of all coins. E.g. if we own the sum of 6 opencent it would be advisable to have coins in the selection of 2,2,1,1 in order to pay all possible amounts. This also prevents *amount tracing*, where an awkward price (13.37) asks for an awkward coin exchange at the issuer beforehand.

So, we need to look at the combined sum of coins received and coins already in possession, and need to find the right coin selection to be able to make all possible future coin transfers. We then need to know which coins to keep, and what blinds to make and paying for the minting using *all* the just received coins and using *some* existing coins.

Also see *Tokenizing* for a sample implementation of a solution.

[1] GNU Taler experiments with this approach: in essence coins don't have serials but keys, which can sign a partial amount to be spent. This requires more smartness to avoid double spending, introducing new problems to be solved.

Fields

- *blinds*: A List of Blinds. *(List of* Blinds*)*
- *coins*: A list of coins. *(List of* Coins*)*
- *message_reference*: Client internal message reference. *(Integer)*
- *transaction_reference*: A random identifier that allows the client to resume a delayed mint/renew process. *(BigInt)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "blinds": [
    {
      "blinded_payload_hash":
↪"7ed0cda1c1b36f544514b12848b8436974b7b9f6c7...",
      "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
      "reference": "b0",
      "type": "blinded payload hash"
    },
    {
      "blinded_payload_hash":
↪"8924dbcf75ab40e3bd3b4d38315722c981fe10946d...",
      "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
      "reference": "b1",
      "type": "blinded payload hash"
    },
    {
      "blinded_payload_hash":
↪"278fc8e4bd861b7206c065004296af57e14963d928...",
      "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
      "reference": "b2",
      "type": "blinded payload hash"
    },
    {
      "blinded_payload_hash":
↪"2995fd1b9e61926d757a516357f9814e20869fe722...",
      "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
      "reference": "b3",
      "type": "blinded payload hash"
    }
  ],
  "coins": [
    {
```

(continues on next page)

(continued from previous page)

```
      "payload": {
        "cdd_location": "https://opencent.org",
        "denomination": 1,
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "mint_key_id": "1ceb977bb531c65f133ab8b0d60862b17369d96",
        "protocol_version": "https://opencoin.org/1.0",
        "serial": "cd613e30d8f16adf91b7584a2265b1f5",
        "type": "payload"
      },
      "signature": "2ec0af339566b19fb9867b491ce58025dcefcab649...",
      "type": "coin"
    },
    {
      "payload": {
        "cdd_location": "https://opencent.org",
        "denomination": 2,
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
        "protocol_version": "https://opencoin.org/1.0",
        "serial": "78e510617311d8a3c2ce6f447ed4d57b",
        "type": "payload"
      },
      "signature": "6aefa7472518ed0a1ec64971220ce3a3a921a70bb0...",
      "type": "coin"
    },
    {
      "payload": {
        "cdd_location": "https://opencent.org",
        "denomination": 5,
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "mint_key_id": "897a16bf12bd9ba474ef7be0e3a53553a7b4ece8",
        "protocol_version": "https://opencoin.org/1.0",
        "serial": "e4b06ce60741c7a87ce42c8218072e8c",
        "type": "payload"
      },
      "signature": "72da93670f666c529f26fcf15092a63c0fa48c8387...",
      "type": "coin"
    }
  ],
  "message_reference": 100004,
  "transaction_reference": "ad45f23d3b1a11df587fd2803bab6c39",
  "type": "request renew"
}
```

Complete example: request_renew.json

3.2.11 ResponseDelay Message

This message is used by the issuer to signal a delay in minting or renewing coins. It is recommended to set up the issuer to be fast and reliable enough to never use this message.

Fields

- *message_reference*: Client internal message reference. *(Integer)*
- *status_code*: The issuer can return a status code, like in HTTP: *(Integer)*
- *status_description*: Description that the issuer passes along with the status_code. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "message_reference": 100004,
  "status_code": 300,
  "status_description": "ok",
  "type": "response delay"
}
```

Complete example: response_delay.json

3.2.12 RequestResume Message

This message request that an action that was delayed before with a *ResponseDelay* is to be resumed. Either a *ResponseMint* is returned, or another ResponseDelay if the client is unlucky.

Fields

- *message_reference*: Client internal message reference. *(Integer)*
- *transaction_reference*: A random identifier that allows the client to resume a delayed mint/renew process. *(BigInt)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "message_reference": 100005,
  "transaction_reference": "ad45f23d3b1a11df587fd2803bab6c39",
  "type": "request resume"
}
```

Complete example: request_resume.json

3.2.13 RequestRedeem Message

This message aks for coins to be redeemed, e.g. converted for real-world money.

The client needs to be authenticated for this request (outside this protocol), so that the issuer knows who to credit the value of the coins to.

Fields

- *coins*: A list of coins. *(List of* Coins*)*
- *message_reference*: Client internal message reference. *(Integer)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "coins": [
    {
      "payload": {
        "cdd_location": "https://opencent.org",
        "denomination": 2,
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
        "protocol_version": "https://opencoin.org/1.0",
        "serial": "cd447e35b8b6d8fe442e3d437204e52d",
        "type": "payload"
      },
      "signature": "11b6bfa18134c300f4440df1db17a08fa71a071b71...",
      "type": "coin"
    },
    {
      "payload": {
        "cdd_location": "https://opencent.org",
        "denomination": 2,
```

(continues on next page)

(continued from previous page)

```
        "issuer_id": "23ed956e629ba35f0002eaf833ea436aea7db5c2",
        "mint_key_id": "f2864e5cd937dbaa4825e73a81062de162143682",
        "protocol_version": "https://opencoin.org/1.0",
        "serial": "5b6e6e307d4bedc51431193e6c3f339",
        "type": "payload"
      },
      "signature": "a6dd7b7f1f12c4e411289e8ea0355f24a8597bbc38...",
      "type": "coin"
    }
  ],
  "message_reference": 100006,
  "type": "request redeem"
}
```

Complete example: request_redeem.json

3.2.14 ResponseRedeem Message

This just answers to a *RequestRedeem*, and doesn't hold other meaningful information.

Fields

- *message_reference*: Client internal message reference. *(Integer)*
- *status_code*: The issuer can return a status code, like in HTTP: *(Integer)*
- *status_description*: Description that the issuer passes along with the status_code. *(String)*
- *type*: String identifying the type of message. *(String)*

Example

```
{
  "message_reference": 100006,
  "status_code": 200,
  "status_description": "ok",
  "type": "response redeem"
}
```

Complete example: response_redeem.json

CHAPTER

FOUR

FIELD REFERENCE

4.1 Field Types

This lists all the field types used in the protocol.

- String: A JSON string.
- Integer: A JSON integer.
- BigInt: A JSON string containing a large number represented as the hex representation of it.
- DateTime: A JSON string containing the ISO representation of a date.
- List: A JSON list that can contain all the possible field types mentioned here.
- URL: A JSON string containing the URL of a resource.
- WeightedURLList: A list of 2 element tuples [Int, URL]. Useful for round-robin, but also reflects a preference. The lower the weight value, the higher the priority.
- Schema / Object: A JSON object that conforms to the given schema.

All fields are mandatory, but can be empty in case of strings.

4.2 Fields

4.2.1 additional_info

A field where the issuer can store additional information about the currency. Free text for humans.

Type: String
Used in: *CDDC*

4.2.2 blind_signature

The signature on the blinded hash of a payload.

Type: BigInt
Used in: *ResponseMint Message*

4.2.3 blind_signatures

A list of BlindSignatures.

Type: List of *BlindSignatures*
Used in: *ResponseMint Message*

4.2.4 blinded_payload_hash

The blinded hash of a payload.

Type: BigInt
Used in: *Blind*

4.2.5 blinds

A List of Blinds.

Type: List of *Blinds*
Used in: *RequestMint Message*, *RequestRenew Message*

4.2.6 cdd

Contains the Currency Description Document (CDD).

Type: *CDD*
Used in: *CDDC*

4.2.7 cdd_expiry_date

The date the CDD expires.

The CDD should not be used or validated after this date.

Type: String
Used in: *CDDC*

4.2.8 cdd_location

Location to download the CDD from.

Useful for clients to "bootstrap" a yet unknown currency.

Type: URL
Used in: *CDDC*, *Payload*

4.2.9 cdd_serial

The version of the CDD.

Should be increased by 1 on a new version.

Type: Int
Used in: *CDDC*, *MKC*, *RequestCDDC Message*, *ResponseCDDSerial Message*

4.2.10 cdd_signing_date

When was the CDD signed?

Type: DateTime
Used in: *CDDC*

4.2.11 cddc

A full Currency Description Document Certificate.

Type: *CDDC*
Used in: *ResponseCDDC Message*

4.2.12 coins

A list of coins.

Type: List of *Coins*
Used in: *CoinStack Message*, *RequestRedeem Message*, *RequestRenew Message*

4.2.13 coins_expiry_date

Coins expire after this date.

Do not use coins after this date.

Type: DateTime
Used in: *MKC*

4.2.14 currency_divisor

Used to express the value in units of ‘currency name’.

Example: a divisor of 100 can be used express cent values for EUR or USD.

Type: Int
Used in: *CDDC*

4.2.15 currency_name

The name of the currency, e.g. Dollar.

Use the name of the ‘full’ unit, and not its fraction, e.g. ‘dollar’ instead of ‘cent’, and use the currency_divisor to express possible fractions.

Type: String
Used in: *CDDC*

4.2.16 denomination

The value of the coin(s).

Type: Int
Used in: *MKC*, *Payload*

4.2.17 denominations

The list of possible denominations.

Should be chosen wisely, so that it allows all possible values within the currency. Should be listed in increasing value.

Type: List of Int
Used in: *CDDC*, *RequestMKCs Message*

4.2.18 id

Identifier, a somewhat redundant hash of the *PublicKey*

This is just a visual helper, and MUST not be relied on. Calculate the hash of the key in the wallet (client software).

Type: BigInt
Used in: *CDD*, *CDDC*, *MintKey*, *MKC*

4.2.19 info_service

A list of locations where more information about the currency can be found.

This refers to human-readable information.

Type: WeightedURLList Used in: *CDDC*

4.2.20 issuer_cipher_suite

Identifier of the cipher suite that is used.

The format is: SIGN-HASH-PADDING-BLINDING, e.g. *RSA-SHA256-PSS-CHAUM82*.

See *cipher suites*.

Type: String
Used in: *CDDC*

4.2.21 issuer_id

The identifier (hash) of the issuer public master key in the CDDC

Type: BigInt
Used in: *MKC*, *Payload*

4.2.22 issuer_public_master_key

The hash of the issuer's public key

The only valid identifier of a currency is the master key.

Type: *PublicKey*
Used in: *CDDC*

4.2.23 keys

A list of Mint Key Certificates

Type: List of *MKCs*
Used in: *ResponseMKCs Message*

4.2.24 message_reference

Client internal message reference.

Set by the client, echoed by the issuer.

Type: Integer
Used in: *RequestCDDSerial Message*, *RequestCDDC Message*, *RequestMint Message*, *RequestMKCs Message*, *RequestRedeem Message*, *RequestRenew Message*, *RequestResume Message*, *ResponseCDDSerial Message*, *ResponseCDDC Message*, *ResponseDelay Message*, *ResponseMint Message*, *ResponseMKCs Message*, *ResponseRedeem Message*

4.2.25 mint_key

The mint key that was signed in the certificate.

Type: *MintKey*
Used in: *MKC*

4.2.26 mint_key_id

Identifier of the mint key used.

Type: BigInt
Used in: *Blind*, *Payload*

4.2.27 mint_key_ids

What mint keys should be returned?

If left emtpy, no filter is applied.

Type: List of BigInt
Used in: *RequestMKCs Message*

4.2.28 mint_service

A list of locations where *Blinds* can be minted into *Coins*

Type: WeightedURLList
Used in: *CDDC*

4.2.29 modulus

The modulus of the public key

Type: BigInt
Used in: *PublicKey*

4.2.30 payload

The payload of the coin.

Type: *Payload*
Used in: *Coin*

4.2.31 protocol_version

The protocol version that was used.

Type: Url
Used in: *CDDC*, *Payload*

4.2.32 public_exponent

The exponent of the public key.

Type: BigInt
Used in: *PublicKey*

4.2.33 public_mint_key

The public key of the mint key.

Type: *PublicKey*
Used in: *MintKey*

4.2.34 redeem_service

A list of locations where *Coins* can be redeemed.

Type: WeightedURLList
Used in: *CDDC*

4.2.35 reference

An identifier that connects *Blind*, *BlindSignature* and blinding secrets.

Set by the client, echoed by the server.

Type: String
Used in: *ResponseMint Message*, *Blind*

4.2.36 renew_service

A list of locations where *Coins* can be renewed.

Type: WeightedURLList
Used in: *CDDC*

4.2.37 serial

The serial of the *Coin*.

This random value is generated by clients. It is used to identify coins and prevent double spending. Once the coin is spent, the serial will be stored by the issuer. It is supposed to be unique for each coin because of its sufficient long length. A high entropy (crypto grade quality) is important.

Type: BigInt
Used in: *Payload*

4.2.38 sign_coins_not_after

Use *MintKey* only before this date.

Type: DateTime
Used in: *MKC*

4.2.39 sign_coins_not_before

Use *MintKey* only after this date.

Type: String
Used in: *MKC*

4.2.40 signature

A signature within a certificate.

Type: String
Used in: *CDDC*, *Coin*, *MKC*

4.2.41 status_code

The issuer can return a status code, like in HTTP:

2XX SUCCESS
3XX DELAY / TEMPORARY ERROR
4XX PERMANENT ERROR

Type: Integer
Used in: *ResponseCDDSerial Message*, *ResponseCDDC Message*, *ResponseDelay Message*, *ResponseMint Message*, *ResponseMKCs Message*, *ResponseRedeem Message*

4.2.42 status_description

Description that the issuer passes along with the status_code.

Type: String
Used in: *ResponseCDDSerial Message*, *ResponseCDDC Message*, *ResponseDelay Message*, *ResponseMint Message*, *ResponseMKCs Message*, *ResponseRedeem Message*

4.2.43 subject

A message that can be passed along with the coin stack.

Can be left empty. Used informally to indicate a reason for payment etc.

Type: String
Used in: *CoinStack Message*

4.2.44 transaction_reference

A random identifier that allows the client to resume a delayed mint/renew process.

This should be a good random number.

Type: BigInt
Used in: *RequestMint Message*, *RequestRenew Message*, *RequestResume Message*

4.2.45 type

String identifying the type of message.

This is the id that is used for parsing the message. One of:

- blinded payload hash
- blind signature
- cdd
- cdd certificate
- coin
- coinstack
- mint key certificate
- mint key
- payload
- rsa public key
- request cddc
- request cdd serial
- request mint key certificates
- request mint
- request redeem
- request renew
- request resume
- response cddc
- response cdd serial
- response delay
- response mint key certificates
- response mint
- response redeem

Type: String
Used in: *ResponseMint Message*, *Blind*, *CDDC*, *CDDC*, *Coin*, *CoinStack Message*, *MKC*, *MKC*, *Payload*, *RequestCDDSerial Message*, *RequestCDDC Message*, *RequestMint Message*, *RequestMKCs Message*, *RequestRedeem Message*, *RequestRenew Message*, *RequestResume Message*, *ResponseCDDSerial Message*, *ResponseCDDC Message*, *ResponseDelay Message*, *ResponseMint Message*, *ResponseMKCs Message*, *ResponseRedeem Message*, *PublicKey*

CHAPTER

FIVE

OPERATIONS

5.1 Cipher suites

We define cipher suites that implementations need to support.

5.1.1 RSA-SHA256-PSS-CHAUM82

This suite uses RSA for the crypto operations. SHA256 is used as a hashing algorithm, and PSS for padding in certificate signatures.

CHAUM82 is used for the blinding, e.g. raw RSA signatures.

The signing process is initialized following the cryptography.io example:

```python
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

private_key = rsa.generate_private_key(65537, 512)
signature = private_key.sign(
    "The json dump",
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
```

See the example rsa suite in rsa_suite.py

5.2 Tokenizing

When preparing blanks we need to decide which values to use. The overall goal is to get the wallet into a state where the next transaction can be made without having to fetch "change" first. E.g. if we have a sum of 200 in the wallet, we want to be able to pay 137 without having fetch change first.

This makes live more comfortable, but more important it helps privacy, because the issuer can't link actions around an awkward price.

We have a sample implementation for this. It contains one line of old black magic though. But at least we have tests for it.

Also see *RequestRenew Message*.

CHAPTER

SIX

APPENDIX

6.1 Scope

We scope the protocol, and this documentation in the following way:

Targeted at developers - developers should be enabled (and motivated) by the OpenCoin protocol to implement standard confirming software components and apps. However, we hope that this documentation is also understandable for the interested user (or founder, investor, auditor, etc.)

Just the protocol - we don't deliver any ready to use implementations. This allows us to fully focus on the protocol, and keeps a separation to actual implementations.

Easy to understand - we try to avoid complexity.[1] This affects the protocol itself as well as its documentation. This means: if you, the reader, don't understand a sentence or a concept, please contact us. We will improve the description. Being easy to understand is one of the main goals of OpenCoin.

Only the core - lots of developments have happened since *we started*. Take the example of messengers like Signal, Telegram or WhatsApp: they have opened new ways to transport messages, and they take care of identifying the communication partner. This especially means that message transport and authentication stay out of scope.

6.2 Downloads

There are no downloadable apps or implementations, but we have little helpers for *you* to implement OpenCoin.

[1] See "The Grug Brained Developer", https://grugbrain.dev/.

6.2.1 Python schemata

We have the schemata in python, using the marshmallow library:

Openoin Python schemata

6.2.2 JSON schemata

We have a version of the OpenCoin schemata translated to JSON Schema:

OpenCoin JSON schemata

6.3 FAQ

6.3.1 Can OpenCoin be abused for [whatever crime]?

People ask if untraceable transactions (like OpenCoin allows) be abused for crimes of all kinds.

WARNING!
FAKE DVDS NEVER HAVE A HAPPY ENDING...
DRUGS
TERRORISM
GUN CRIME
CHILD TRAFFICKING
Buying fake DVDs funds local crime
ISLINGTON

Fig. 1: Copying DVDs also finances evil crimes*Found in a bus stop in Islington*

The usual candidates for crimes are terror financing, child pornography, money laundering, blackmail etc. These are all evil crimes, without doubt. Untraceable transactions would facilitate these crimes because a payment could be made to an unknown wallet/client, which would renew those coins, and later on distribute the coins to other wallets, which would then convert the coins to "real money".

We think that this risk is quite real. However, the question is if tracing money transactions is the (one) way to stop the crime. Tracing transactions has a huge impact on the privacy of innocent people. So there is a tradeoff. How has society decided for other fields? E.g. streets, the postal service, knives and the internet can be abused as well, but generally speaking, societies don't trace buying or using those things. Some would even say that one of the main uses of the 500 EUR banknote is dubious or illegal activity, and it still exists.

Our point is that yes, untraceable payments can be used for crimes, but so can other services as well, and it is up for society to decide how to handle it.

6.3.2 What about taxation?

One could ask if untraceable payments can be used to avoid taxation, or the other way around: should a mechanism be built into a transaction schema to enforce taxation[1].

First: we support taxes, they are needed for a society to work.

If one wanted to help the tax office here, one could think about requiring authorization for renewing coins as well as for minting and redeeming them. By this all points of contact with the issuer could be traced, and one could at least follow the amount of coins flowing through a wallet. It would be up for legislation to decide what actions (mint, renew, redeem) to tax in what way.

We find however that in the "real world" taxation is enforced using a different mechanism: receipts. Warranty and the ability of the customer to deduct costs for tax purposes is bound to having proper receipts. So the recipient is forced py the payer to produce a receipt, and make the transaction official. This seems to work quite fine, and is suitable for electronic payments as well as cash payments. We think that this is the right place to handle these issues.

6.3.3 Can I legally use OpenCoin for xyz?

You have to talk with your lawyer to check the current situation for country and purpose. The old *legal report* can be a starting point.

6.3.4 Can OpenCoin be used offline?

When receiving a coin, the coin needs to be *renewed*. This is needed to prevent the sender from double spending the coin. To reach the issuer you will most likely to be online. However, if you trust the center, you could delay the renewal operation as long as you like. The magic of trust…

6.3.5 OpenCoin and ripple?

The OpenCoin project that started in 2007 to create an open source protocol for the the electronic cash system invented by David Chaum. It is about minting tokens that can be transferred in a non-traceable way. It is fast, and the user has full flexibility on how to do the transfer.

It had nothing to do with "OpenCoin Inc.", a younger company that was developing the ripple network. Unfortunately they decided in 2012 to name their company the same as our project. That created a bit of confusion, as both are about electronic transfers.

The name of the company than changed to ripplelabs, and they gave the domain name back to us, the OpenCoin project. For free. How cool is that?

[1] see the section on GNU Taler in the *overview*.

6.4 Reviews

For the first version of OpenCoin we had funding from the LDA, which allowed us to get OpenCoin reviewed from a technical and legal perspective.

6.4.1 Technical / Crypto review

The review of the preliminary protocol was done by:

- Alex Dent
- Kenny Paterson
- Peter Wild

The technical review did not find "any obvious flaws".[1]

6.4.2 Legal review

We also had a legal review of our ideas by:

- Chris Reed
- George Walker

The legal report examines under which circumstances OpenCoin could be run (at the time). Legal circumstances have changed since then, so we would recommend do have another legal review.

There was also a freemind mindmap, that shows the structure of the legal issues. Here exported as an image:

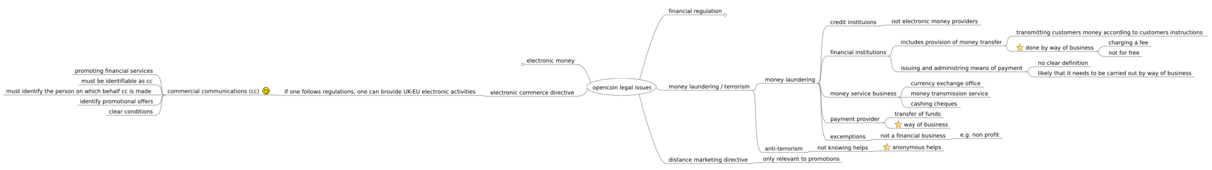

promoting financial services
must be identifiable as cc
must identify the person on which behalf cc is made
identify promotional offers
clear conditions
commercial communications (cc)
if one follows regulations, one can brovide UK-EU electronic activities
electronic commerce directive
electronic money
opencoin legal issues
financial regulation
money laundering / terrorism
money laundering
credit instituions
not electronic money providers
financial institutions
includes provision of money transfer
transmitting customers money according to customers instructions
done by way of business
charging a fee
not for free
issuing and administring means of payment
no clear definition
likely that it needs to be carried out by way of business
money service business
currency exchange office
money transmission service
cashing cheques
payment provider
transfer of funds
way of business
excemptions
not a financial business
e.g. non profit
anti-terrorism
not knowing helps
anonymous helps
distance marketing directive
only relevant to promotions

Fig. 2: Legal issues around OpenCoin

It is also available as a standalone html file

[1] "We did not fall of our chairs, laughing, which we normally do, when we are approached by industry"

6.5 History

6.5.1 v0.1

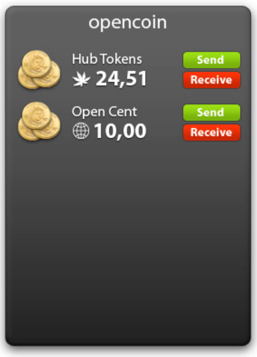
We created the initial version of the protocol.

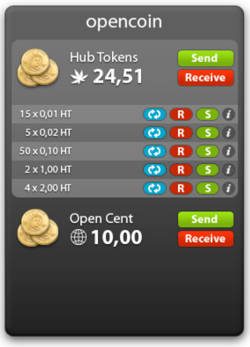
On top of it we built a proof of concept (PoC) to show that the whole system actually works. The PoC worked, but would be implemented differently today.

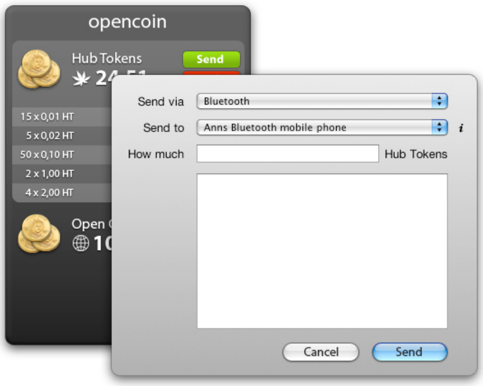
The protocol however stayed more or less the same compared with today's version.

GUI designs

Some gui designs were made:







6.5.2 v0.2

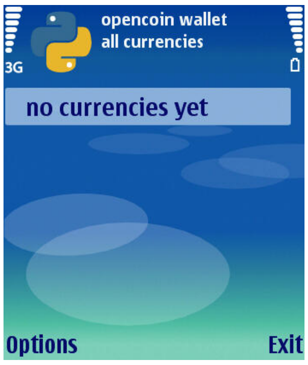
Approaches to improve OpenCoin.

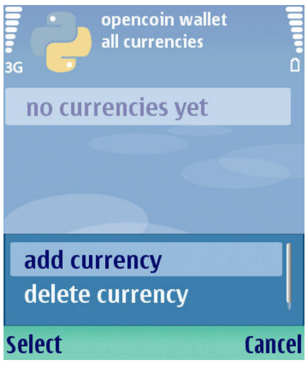
Extended protocol

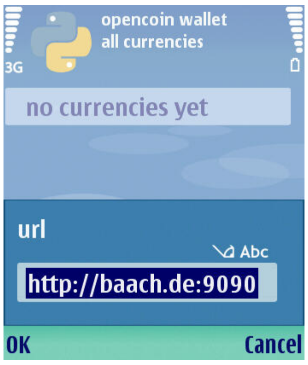
The protocol was extended: https://github.com/OpenCoin/opencoin-historic/tree/master/standards

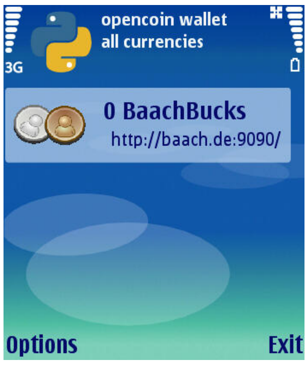
pys60 client

This client runs on s60, which is the operating system that was used by early Nokia smartphones. There was a python version for s60 (called pys60) that allowed running python scripts.









python version

This was a python version to be run for documentation purposes:

- https://github.com/OpenCoin/opencoin-historic/tree/master/sandbox/jhb/oc2

6.5.3 v0.3

An attempt to bring the protocol closer to its origins, and to create a pure javascript implementation.

6.5.4 v0.4

This is the current version, where the focus is on the OpenCoin protocol only. We think now that just creating the protocol, and keeping it as small and simple as possible is the *best approach* for implementers.

6.6 Ideas

These are some ideas that might be implemented in the future

6.6.1 .oc file suffix

It might make sense to have a file ending for OpenCoin messages, especially for *CoinStacks*. This would allow mime-handlers to pickup the file and treat them in a special way.

6.6.2 oc over html

In theory a *CoinStack* could come as a standalone mini-wallet that would display the contents of the embedded CoinStack in a readable way, along with ways to renew the coins etc.

The problem is that this would make users open up and trust arbitrary html files.

6.6.3 opencoin.net web wallet

We could implement a web based wallet (using javascript or python in the browser (pyscript/transcrypt). This wallet could be either a single html file, containing all css and js, or we use subresource integrity for this.

This web interface could even be the hander for *.oc files*.