# Usernames for I/OCoin


## Abstract

Nobody likes using long, Base58-encoded string to transfer their money; their seemingly random structure makes it extremely difficult to quickly distinguish between different destinations. Our goal is to create a system which can be directly implemented into I/OCoin's normal operations, where arbitrary strings, henceforth referred to simply as "usernames", can be easily associated with Base58-encoded addresses. This will allow for a user to associate their I/OCoin address to a simplified string at a specific point on the blockchain, after which every other node connected to the I/OCoin network will be able to send coins to the simplified string, which will be resolved to the complicated Base58-encoded address.


## Contents



## 1 Motivation

Currently, in order to send coins to another person, you need to use their Base58-encoded address. These addresses range from 27-34 characters in length, and consist of a mostly random sequence of alphanumeric characters, which puts a limit on the methods that can be used to transmit this address to other parties due to their complexity. Various methods have been devised to attempt to deal with this, such as QR codes or the "Address Book" built into each coin's Qt wallet, but each of these has their limitations. A QR code is simply converted into the Base58 encoded address, meaning that an interested party would still need to either store the address or QR code somewhere for future reference, and each "Address Book" instance is privately maintained by each wallet, that is, the address associations stored in the address book are not shared with any other nodes in the network.

## 2 Protocol

In order to support this system, we have devised a protocol consisting of the minimal operations required for its implementation. These operations consist of the everything that is needed to associate a username with an address, remove an association, and resolve conflicts between associations. It is assumed that the desired method to use will be transmitted along with any other data involved.

### 2.1 Requests

These are used by a client to either claim, or revoke their claim to a username for an address. They will usually result in the creation of a transaction on the blockchain. Each of these methods initiates communication with the entity that processes association requests, and will result in either a successful request, or an unsuccessful request.

#### 2.1.1 Username Association

This is the method that is used by the owner of an address to attempt to claim a username. The data that needs to be sent is a proof-of-ownership of the address, along with the desired username.

#### 2.1.2 Username Association Removal

This is the method that is used by the owner of an address and associated username to revoke their claim on the username. If only one username is allowed per address, only the proof-of-ownership of the address is required, otherwise, the username is also required.

### 2.2 Conflict Resolution

In the simplest case, a client will request a username that has not yet been claimed, and we can simply assign them that address. However, in the case that two clients attempt to claim the same username, there needs to be a way to determine who should get the username. Our recommended solution is to require that all requests be initiated via a transaction on the blockchain. This allows easy selection of the winner, as we can simply select the client's transaction with the smallest block height, or earliest transaction in the case of equal block heights. In the case of completely equal times, you can simply randomly select one of the clients to be the winner.

It should be noted that the chances of this event occurring change depending on the implementation method; in some cases, it may be possible to completely avoid any chance of conflicts.

### 2.3 Retrievals

This is the core method that is used by every client when attempting to lookup a username. For this reason, it will need to be implemented efficiently to support many clients simultaneously attempting to resolve usernames. The minimal version would simply return either an address if there is an association for the username, or some form of `null` if there is not an association. A more complicated implementation could choose to return further information such as "pending" in the case that a client has applied for the username but it is not yet safe to send to the address.

## 3 Implementation Options

There are primarily three different methods that can be used to implement this system, each differing in the amount of centralization they entail. These methods will be discussed further in the below subsections. No matter which method is implemented, functionality will be directly implemented into the client with support for:

- Displaying the username currently associated to any addresses you own
- The ability to manually import and address into your local "Address Book"
- Searching for an address using a username
- Sending directly to a username
- Full backwards compatibility (can still send directly to an address)
- Optionally, reverse lookup of a username from an address
- Optionally, the ability to browse the entire address book

## 3.1 Fully Centralized

In the fully centralized option, there will be a single, dedicated address-resolving server; an IONS (I/OCoin Name Server).

The implementation of this option will require some form of key-value store for the username-to-address associations, along with a system for handling the above described protocol. We assume that the server will have a web interface and API which allows access to all important functionality described below.

We will start by describing the process that would occur for a user attempting to register for a username for their address.

1. A client selects the username they would like and submits it to the server.
2. The server receives the request, and checks if the username has already been claimed.
3. If it has been claimed, a message is returned to the client to indicate this.
4. Otherwise, a new I/OCoin "Payment Address" is created via the `iocoind` daemon and associated to the requesting address in a database of pending subscriptions.
5. This "Payment Address", along with an optional fee and timeout, is sent back to the client.
6. The client is provided with this information, and in the case they are using the Qt client, a button will be provided to simplify accepting.
7. If the client still wants the username for the displayed fee, they can send the required fee to the supplied address, otherwise, their request will time out and other clients on the network will be able to request the username.
8. Once the server has received the fee with a sufficient amount of confirmations, the username-to-address association is stored in the database, and is available for further lookups.

### 3.1.1 Benefits

The major benefits of this option are its simplicity, and resistance to attacks based on forking the blockchain. If a fork were to occur after an association was established, the association would persist. Although the fee would be lost, there is no risk of a blockchain attack which is able to modify the endpoint of a username.

## 3.2 Partially-Centralized Deterministic

This option consists mostly of the previous option, but stores some of the data on the blockchain. This allows any party to reconstruct the username-to-address mapping using a deterministic algorithm with the blockchain as an input. However, to be fully deterministic, a more robust method than random selection would need to be defined for selecting the winner in the case of a conflict. A possible solution will be provided later.

For this option, there would still be a centralized resolver server, but confidence in the server would be greatly increased by publicly storing data in the blockchain; this method would be similar to a partial integration of Namecoin into I/OCoin. In the minimal case, any claims and releases of claims to usernames, along with the associated address, would need to be stored somehow in the blockchain.

The simplest implementation is to create new transaction types for each of the methods, where the `version` of the transaction is used to indicate the method; either `claim` or `release`; and a new `username` field is used to store which username the operation is to be performed on. The address can be derived from the `vins` of the transaction, or other parts of the transaction.

An important thing to note is that if the fully centralized option was already in place, it is trivial to migrate to this version. All that would be required is a dump of the current username-to-address mapping for bootstrapping, and for clients to switch over to the new wallet. The new transaction types would be added to the blockchain, and provided enough testing is performed beforehand, the switch can be seamless.

### 3.2.1 Full Determinism

In order for this method to be fully deterministic, it would require a recommended or reference implementation of a deterministic algorithm for deriving the association table from the initial bootstrap table and

blockchain. If a resolver wants to be trusted, they would also be required to provide a deterministic method that they use to resolve conflicts.

A possible version of the deterministic algorithm would be performed on a block-by-block basis, and could involve:

- For each block:
  - Collect the transactions in the block which are `request` or `release` transactions.
  - Sort the transactions first by `txid.time`, then by `hash(txid)`.
  - For each transaction:
    - If the transaction is a `request` and the requested `username` is free, grant the request, otherwise, deny it.
    - If the transaction is a `release` and the requested `username` is claimed, release the address.
  - Commit the changes to the database

The above algorithm's determinism is based on the determinism of the `hash(txid)` function; it should be selected so that it provides a random value which cannot be easily minimized, that is, it is difficult for a third-party to create a `txid` that is guaranteed to return a very small value for `hash(txid)`.

#### 3.2.2 Benefits

Most of the benefits are from the increased decentralization, but there are also some benefits to the resolver server:

- **More Decentralized** – There is no longer a single entity able to control the associations or decide on how conflicts are resolved. Other parties are free to create and use alternate servers if they are not satisfied with the official ones.
- **Failure Resistant** – Even if the resolver server is completely compromised, and there are no backups, as long as somebody knows how conflicts were resolved and the method was deterministic, the full association table can be reconstructed using the above described algorithm.

#### 3.2.3 Drawbacks

The introduction of these new transaction types which contain information unrelated to the actual monetary transactions introduces some possible issues:

- **Blockchain Bloat** – The major issue with moving towards a decentralized solution is the increase in the blockchain size required to store the additional data. Assuming a maximum username of 14 characters, each a single byte, and 2 bytes left for other parts of the transaction, each request will be approximately 16 bytes. Assuming that people will prefer to hold a username rather than release it, this would result in a blockchain size increase of approximately 16 MB/1000 usernames. Using compression, this number may be further reduced.
- **Forks** – With this options, forks to the blockchain can be extremely dangerous in the case that the blockchain is not constantly monitored. With proper planning, somebody could wait for an address to claim a username and share it, then fork the blockchain before the claim went through and claim the target username to their address instead, effectively "stealing" the username. This is actually a case when having a centralized server is beneficial; they can ensure that conflicts are resolved in a manner such that nobody is able to steal and address.

It should be noted that there is another way that forks can be dealt with; by enabling strict sync-checkpoint checking, a node will disable itself in the case that it receives a sync-checkpoint conflict, which could occur in the case of an attack. If the additional constraint is added that the wallet be disabled if no sync-checkpoint has been received within a certain block height window, then a confirmation limit could be selected for username request transaction that would guarantee a fork would not affect the transaction. In summary, fork-related issues can be avoided as long as a node is overly-strict when dealing with sync-checkpoints.

### 3.3 Fully Decentralized

In this option, the address-to-username association table would be fully decentralized, meaning that there would no longer be a resolver server. The naïve solution is to simply require that every client store it's own

copy of the address book, which it derives using the above described deterministic algorithm. The main drawback of this approach is that each association entry would consist of at most 34 bytes for the address, and 14 bytes for the username, for a total of 48 bytes/entry. For 1000 entries, this would result in a 48 MB table. For comparison, BlackCoin recently had an average of 5000 active addresses/day, which would result in a 250 MB table, which is almost as large as the blockchain itself.

The more elegant solution is integrate a Distributed Hash Table (DHT) implementation directly into the client, which would allow each client to have quick access to the address book, and only need to store a small part of it. The most obvious drawback of this method is that every client would need a way to quickly ensure that the DHT that is currently being broadcast is valid, that is, validating the integrity of the DHT. This is usually a difficult problem to solve for arbitrary DHT tables, but since we have a deterministic algorithm capable of deriving the entire DHT table, it is likely that a method could be created capable of performing this validation.

#### 3.3.1 Benefits

The main benefit is that this option is fully decentralized; there is no central authority responsible for controlling all the username associations.

## 4 Conclusion

The above should provide a fairly complete summary of the options available for implementing usernames for I/OCoin. The best solution is probably to implement the usernames in phases. Start with the fully centralized version, so the usernames are available for use while development of the partially centralized version proceeds on testnet. Once this has completed and been integrated into the mainnet blockchain, development can begin on the fully decentralized version.