Beerchain

Creating the beer-based cryptocurrency

Yellowpaper version 0.3, 3/11/2018

Beerchain Technology UG (haftungsbeschränkt)
August-Riedel-Str. 9
95447 Bayreuth
Bavaria
Germany

https://www.beerchain.technology
contact@beerchain.technology

Tax ID: DE 315887439
D-U-N-S number: 314910152

Disclaimer: This draft contains information on technical ideas and implementations for the interested reader. It will be greatly extended as development continues.

Content

Concepts

It is assumed that you have read the whitepaper. If you have not, please go to our website (https://www.beerchain.technology) and read it. This section builds on the general concepts in technical means.

Codes

Codes can be placed in various places in various formats, and the most obvious und most human-readable format would be numbers. However, as we need to create 20,800,000,000 codes in the long run and because for security reasons we cannot just use exactly that amount of numbers but rather a magnitude more, the code would be to long for humans to comfortably read and type. Of course, human-readable codes are not our primary intention and the user would not care if the code was represented by a QR or NFC system. However, to avoid future problems, the code should be human-friendly.

To shorten the codes, characters other than numbers can also be used. There already exists a standard called base64 (https://en.wikipedia.org/wiki/Base64), which is suited for our codes. Multiples of 4-character blocks can be used to represent 24 bits per block. While the minimum of 4 characters would lead to a too little number of 16,777,216 codes, 8 characters, i.e. 48 bits, provide 281,474,976,710,656 different numbers. This is also more than 10,000 times more codes than we need, which is great to make brute force attacks more difficult. Codes from this space of possibilities will be randomly taken, there is no pattern.

As the standard base64 encoding is problematic if used in URLs, we use “base64url” instead. The character “+” is replaced by “-“, and “/” is replaced by “_”.

To keep special codes for demonstration and information purposes out of the regular space, those codes are built with an additional leading bit, i.e. in practice with another 4 characters. As those codes are not intended to be read or typed by regular users, the length should be no problem.

Currently, those 8-character codes are intended to be encoded in QR codes, to be written on NFC tags, and to be printed on shop checks.

The codes used in the demonstration app are:

Bronze:	AAABAAAAAAAB	(0x000001000000000001)
Silver:	AAABAAAAAAAK	(0x00000100000000000A)
Gold:	AAABAAAAAABk	(0x000001000000000064)
Diamond:	AAABAAAAACcQ	(0x000001000000002710)

Contract

This section goes step by step through the Beercoin smart contract that has already been deployed. The full code can be found at the end of this document or on GitHub.

InternalBeercoin

The InternalBeercoin classes defines basic constants on coin values as well as a variable representing the amount of bottle caps that can still be produced.

There are four kinds of bottle caps that can be produced and scanned, with no amount exceeding the maximum of a 64-bit value. Because of that, all four values are stored in a single 256-bit variable. “Batch” additions and subtractions are easily possible this way and much gas can be saved as the amount of state changes can be reduced. “Batch” comparisons (e.g. scanned < produced) are possible as well.

A last variable tracks the amount of burnt Beercoins.

ExplorableBeercoin

As the previously defined fields are not accessible from outside, getter functions exist to not only reveal values but also to parse the not understandable 4-in-1 variables. To extract the relevant bits, shifts and masks are used. Some convenience functions for total amounts are also provided.

ERC20Beercoin

This contract defines the functions from the ERC20 standard. Transfers cannot be made to the null address, as a burn function will be introduced later. To avoid redundancy and state changes, the total supply is no independent variable but an on-demand calculation using the initial and burnt supplies as well as the scanned caps.

MasteredBeercoin

A “master” (contract owner) is defined here who has special capabilities. These are batch debit and credit functions that are used to reduce the gas-requiring overhead of individual transactions to and from users. Direct debiting is only possible if a user allowed it first.

Beercoin

The core functionality of the project. The produce method takes the number of caps ordered by a brewery and calculates the amounts of cap types. The scan method takes users and caps to add respective amounts of Beercoins to the balances. The burn method takes the amount to be burnt. All three methods can only be called by the owner, which is our company.

Server

The project presents some unique design challenges that need to be accounted for. In this document several Designs will be presented and discussed.

Technology

The Server will run node.js and make use of the web3.js library, the transactions are signed on a system not connected to the internet.

Concepts

Registration

As we expect that most of our users initially don't know much about the blockchain concept, a log-in via social media is prosed when starting the app for the first time. On the server side this means that an Ethereum address for the new user is created and the private key and password are securely stored. Additionally, the server executes one and only one transaction in the new user's name, that is allowing direct debiting, so we can fully interact with the address via our contract functions. We also charge that new account with Ether for this transaction. After that, we do not touch the private key and password again.

Scanning

When a user scans a code that is sent to the server, it needs to be verified there. This means that the code must be present in the database table of unscanned codes and the request source must be our app. If neither holds true, an attack is assumed, and the issue is investigated.

If code and source are valid, the address of the sending user will be added with the respective amount to a pool of transactions to be made. The user will not immediately receive Beercoins, although the user interface will show new coins. As we expect multiple users to scan multiple coins during an evening, we collect scans before executing a blockchain transaction to save on gas.

Redemption

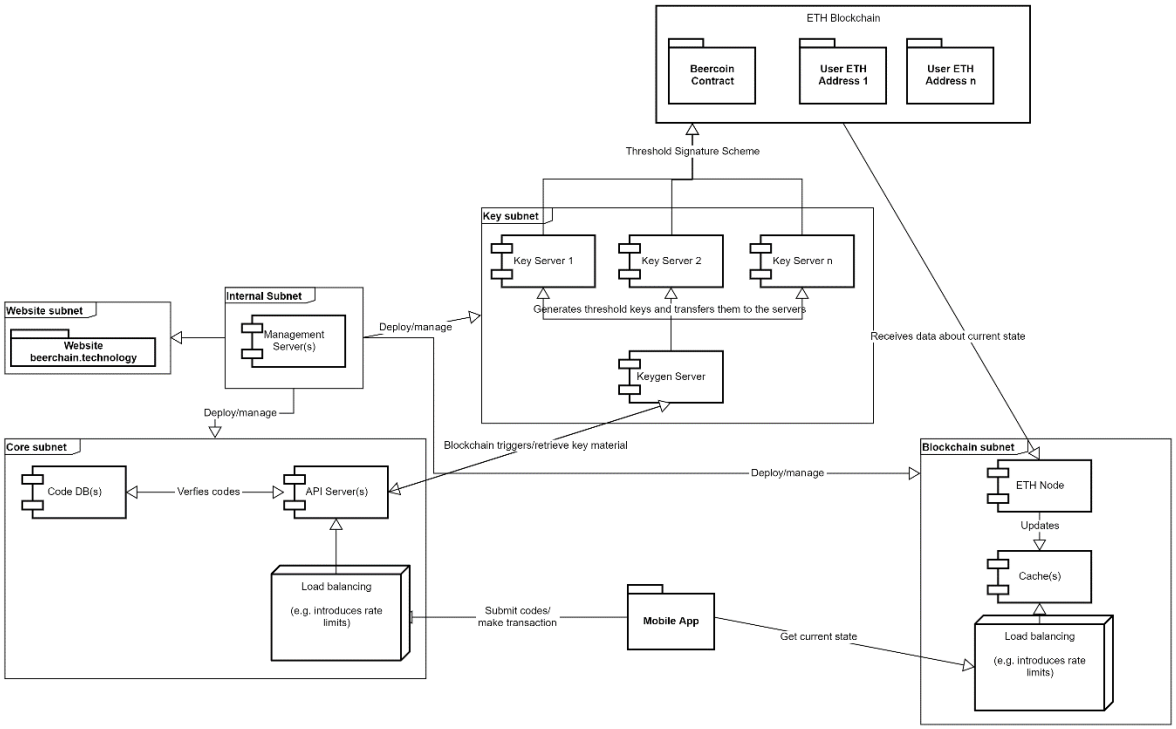
Beercoins can be given to charity and breweries. To easily do so from within the app, the server withdraws Beercoins from the user via direct debiting and sends them to the receiver. User managing their keys themselves can also send Beercoins themselves to our address and we will redirect them appropriately.

Assumptions

- Private keys of all users (except the ones who opt-out) are managed by us.
- Server(s) for accepting and validating codes is needed
- Server(s) for Website hosting are needed
- Ethereum contract only accepts transactions from Beerchain UG controlled addresses.
- Components are modular

The general infrastructure is relatively simple, however for the important components such as the private keys, different solutions can be implemented.

The general infrastructure can be explained on a higher level like this:



Infrastructure

In this chapter the components of the infrastructure will be explained.

Networks

There are several different networks which are all to be understood as separated from each other. Communication must only be possible between specific components and they must be separated by e.g. listening only on the respective networks.

Core

The core subnet contains the core components that are necessary for the conversion of codes to coins and the creation of new codes from breweries. Load balancers ensure the load is equally distributed between the API Servers.

Connection list:

ConnectionID	Direction	From/To	Comment
APP1	In	Mobile App	Submit code or make a transaction
KEY1	Out	Key subnet	Trigger coin transfer to Blockchain
MGMNT1	In	Internal subnet	Deploy or manage (e.g. add new codes)

Internal

This subnet must only be used for managing all other subnets and has additional security requirements. Access to this subnet must be highly secured and abuse must be prevented. All management and deployment processes are carried out via one or several jumphosts which must not accept incoming connections from any subnet and only from authenticated SSH keys and trusted locations (e.g. IP-Ranges).

Key

In the Key subnet all key material is stored, therefore high security requirements need to be met. Depending on the used Design (see in the following chapters), the detailed components may differ. The general idea is to have incoming connections from the Core and Internal subnet.

Blockchain

Here, all components that are used to access the blockchain in a reading manner are located. The concept here is to have a read only access to the blockchain to allow the users to review their current wallet status.

Website

Only used for hosting the website and its components. This must be strongly separated from all the other subnets (except management).

Components

Generally speaking, all components accept connections from the internal subnet as it is used for management and backup purposes.

API Server

Generally speaking, this server is primarily used for sanitizing and normalizing the input that comes from the user. The data is then handed over to other components (e.g. Keygen Server) in a normalized format. It is primarily used to receive and validate incoming codes from the app. The codes are stored in a code database. Additionally, it will trigger transactions on behalf of the user.

ConnectionID	Direction	From/To	Comment
IO_LB_CORE	In/Out	Load Balancer	App connections through LB
IO_DB	In/Out	Code DB Server(s)	Database communication
O_KEY_API	Out	Keygen Server	Blockchain actions on behalf of the user

Code DB

Contains all the codes that are valid for coin redemption.

Keygen Server

This server is mainly used for creation and handing over messages that need to be broadcasted to the Ethereum Blockchain. Upon user request, a key can be exported from here, too. This means that this server needs to be able to authenticate a user.

ConnectionID	Direction	From/To	Comment
IO_KEY_API	In/Out	API Server	Key creation/ Blockchain manifestation trigger
IO_MGMNT	In/Out	Internal subnet	Backup and management
O_APP_KEY	Out	Mobile App	Retrieve key material from our servers.

Key Server(s)

These server(s) hold the private key(s) needed for communicating with the smart contract and the generated addresses. The key material is split between them. How this happens is detailed in the Design options below. Generally speaking, a threshold signature scheme must be used to ensure a compromise between security and availability.

ConnectionID	Direction	From/To	Comment
IO_KEYGEN	In/Out	Keygen Server	Receive messages to sign or new key material
IO_MGMNT	In/Out	Internal subnet	Backup and management
	Out	Mobile App	Retrieve key material from our servers.

Cache (Blockchain subnet)

The cache holds eventually updated state information about the Ethereum blockchain. It is a static mirror of the data retrieved via the Ethereum Node. The data can then easily be requested from mobile apps.

ConnectionID	Direction	From/To	Comment
IO_LB_BC	In/Out	Load Balancer Blockchain subnet	Incoming requests from the app.
I_ETHNODE	In	ETH Node	Update important data in the cache
IO_MGMNT	In/Out	Internal subnet	Backup and management

ETH Node

This is a (full) node that automatically compiles ETH addresses that have been accessed in the past and creates updated data for the caches. In case of a cache miss, the data is retrieved on demand.

Design 1 – MultiSig delegated contract

This design builds on using one contract that the wallet servers[1] interact with, where a certain threshold in consensus must be reached for an action to happen. This means that a minimum of X servers must sign a transaction for the "MultiSig delegated contract" to accept and act upon it.

Advantages

- Clear use separation can be achieved. (e.g. addresses A and B are only allowed to trigger action X, whereas Address C together with D are only allowed to trigger action Y)
- Ensures clean interface to smart contract functions

Disadvantages

- Increases gas cost
- Harder to maintain

[1] https://github.com/christianlundkvist/simple-multisig/blob/master/contracts/SimpleMultiSig.sol

Design 2 – ECDSA Threshold Signature Scheme

Based on:

- https://eprint.iacr.org/2016/013.pdf
- https://www.cs.princeton.edu/~stevenag/threshold_sigs.pdf
- https://cointelegraph.com/news/threshold-signatures-are-a-significant-milestone-in-bitcoin-security (Usecase example)

The idea is to spread the key material on the Key servers in such a way that no server on its own can reproduce the full key. Instead each server must be queried to sign a given message on its own.

Advantages

- Signing and keys are spread locally
- Does not require a smart contract
- Cheaper and easier to maintain (as code is on premise)

Disadvantages

- Use separation cannot be implemented

App

The app serves as the central access point to our service.

Technology

The app is developed with Xamarin for easy maintenance and deployment to Android, iOS, and Windows. The following packages are used:

- ZXing for cross-platform QR code scanning
- Geolocator for cross-platform GPS usage
- Nethereum for the blockchain integration
- Platform-specific SDKs for AdMob and Microsoft advertising

Design

The app design follows the Model-View-ViewModel (MVVM) pattern.

Distribution

There will be two versions of the app:

- The standard app for daily use
- A version for demonstration purposes that only reads special demonstration codes and does not write any data to the blockchain. Development already finished.

Both versions are built from the same codebase, reducing maintenance efforts.

Gaming ecosystem

Besides the app, a whole ecosystem of Beercoin-based games is possible. Following the Xamarin approach, those games will be developed using the Unity engine to easily deploy to a variety of platforms.

Code

```
pragma solidity ^0.4.19;


/**
 * A contract containing the fundamental state variables of the Beercoin
 */
contract InternalBeercoin {
    // As 18 decimal places will be used, the constants are multiplied by 10^18
    uint256 internal constant INITIAL_SUPPLY = 15496000000 * 10**18;
    uint256 internal constant DIAMOND_VALUE = 10000 * 10**18;
    uint256 internal constant GOLD_VALUE = 100 * 10**18;
    uint256 internal constant SILVER_VALUE = 10 * 10**18;
    uint256 internal constant BRONZE_VALUE = 1 * 10**18;

    // In addition to the initial total supply of 15496000000 Beercoins,
    // more Beercoins will only be added by scanning bottle caps.
    // 20800000000 bottle caps will be eventually produced.
    //
    // Within 10000 bottle caps,
    // 1 (i.e. every 10000th cap in total) has a value of 10000 ("Diamond") Beercoins,
    // 9 (i.e. every 1000th cap in total) have a value of 100 ("Gold") Beercoins,
    // 990 (i.e. every 10th cap in total) have a value of 10 ("Silver") Beercoins,
    // 9000 (i.e. every remaining cap) have a value of 1 ("Bronze") Beercoin.
    //
    // Therefore one bottle cap has an average Beercoin value of
    // (1 * 10000 + 9 * 100 + 990 * 10 + 9000 * 1) / 10000 = 2.98.
    //
    // This means the total Beercoin value of all bottle caps that will
    // be eventually produced equals 20800000000 * 2.98 = 61984000000.
    uint64 internal producibleCaps = 20800000000;

    // The  amounts of diamond, gold, silver, and bronze caps are stored
    // as a single 256-bit value divided into four sections of 64 bits.
    //
    // Bits 255 to 192 are used for the amount of diamond caps,
    // bits 191 to 128 are used for the amount of gold caps,
    // bits 127 to 64 are used for the amount of silver caps,
    // bits 63 to 0 are used for the amount of bronze caps.
    //
    // For example, the following numbers represent a single cap of a certain type:
    // 0x0000000000000001000000000000000000000000000000000000000000000000 (diamond)
    // 0x0000000000000000000000000000000100000000000000000000000000000000 (gold)
    // 0x0000000000000000000000000000000000000000000000010000000000000000 (silver)
    // 0x0000000000000000000000000000000000000000000000000000000000000001 (bronze)
    uint256 internal packedProducedCaps = 0;
    uint256 internal packedScannedCaps = 0;

    // The amount of irreversibly burnt Beercoins
```

```
    uint256 internal burntValue = 0;
}

/**
 * A contract containing functions to understand the packed low-level data
 */
contract ExplorableBeercoin is InternalBeercoin {
    /**
     * The amount of caps that can still be produced
     */
    function unproducedCaps() public view returns (uint64) {
        return producibleCaps;
    }

    /**
     * The amount of caps that is produced but not yet scanned
     */
    function unscannedCaps() public view returns (uint64) {
        uint256 caps = packedProducedCaps - packedScannedCaps;
        uint64 amount = uint64(caps >> 192);
        amount += uint64(caps >> 128);
        amount += uint64(caps >> 64);
        amount += uint64(caps);
        return amount;
    }

    /**
     * The amount of all caps produced so far
     */
    function producedCaps() public view returns (uint64) {
        uint256 caps = packedProducedCaps;
        uint64 amount = uint64(caps >> 192);
        amount += uint64(caps >> 128);
        amount += uint64(caps >> 64);
        amount += uint64(caps);
        return amount;
    }

    /**
     * The amount of all caps scanned so far
     */
    function scannedCaps() public view returns (uint64) {
        uint256 caps = packedScannedCaps;
        uint64 amount = uint64(caps >> 192);
        amount += uint64(caps >> 128);
        amount += uint64(caps >> 64);
        amount += uint64(caps);
        return amount;
    }
```

```
/**
 * The amount of diamond caps produced so far
 */
function producedDiamondCaps() public view returns (uint64) {
    return uint64(packedProducedCaps >> 192);
}

/**
 * The amount of diamond caps scanned so far
 */
function scannedDiamondCaps() public view returns (uint64) {
    return uint64(packedScannedCaps >> 192);
}

/**
 * The amount of gold caps produced so far
 */
function producedGoldCaps() public view returns (uint64) {
    return uint64(packedProducedCaps >> 128);
}

/**
 * The amount of gold caps scanned so far
 */
function scannedGoldCaps() public view returns (uint64) {
    return uint64(packedScannedCaps >> 128);
}

/**
 * The amount of silver caps produced so far
 */
function producedSilverCaps() public view returns (uint64) {
    return uint64(packedProducedCaps >> 64);
}

/**
 * The amount of silver caps scanned so far
 */
function scannedSilverCaps() public view returns (uint64) {
    return uint64(packedScannedCaps >> 64);
}

/**
 * The amount of bronze caps produced so far
 */
function producedBronzeCaps() public view returns (uint64) {
    return uint64(packedProducedCaps);
}
```

```
    /**
     * The amount of bronze caps scanned so far
     */
    function scannedBronzeCaps() public view returns (uint64) {
        return uint64(packedScannedCaps);
    }
}

/**
 * A contract implementing all standard ERC20 functionality for the Beercoin
 */
contract ERC20Beercoin is ExplorableBeercoin {
    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);

    mapping (address => uint256) internal balances;
    mapping (address => mapping (address => uint256)) internal allowances;

    /**
     * Beercoin's name
     */
    function name() public pure returns (string) {
        return "Beercoin";
    }

    /**
     * Beercoin's symbol
     */
    function symbol() public pure returns (string) {
        return "🍺";
    }

    /**
     * Beercoin's decimal places
     */
    function decimals() public pure returns (uint8) {
        return 18;
    }

    /**
     * The current total supply of Beercoins
     */
    function totalSupply() public view returns (uint256) {
        uint256 caps = packedScannedCaps;
        uint256 supply = INITIAL_SUPPLY;
        supply += (caps >> 192) * DIAMOND_VALUE;
        supply += ((caps >> 128) & 0xFFFFFFFFFFFFFFFF) * GOLD_VALUE;
        supply += ((caps >> 64) & 0xFFFFFFFFFFFFFFFF) * SILVER_VALUE;
```

```
    supply += (caps & 0xFFFFFFFFFFFFFFF) * BRONZE_VALUE;
    return supply - burntValue;
}

/**
 * Check the balance of a Beercoin user
 *
 * @param _owner the user to check
 */
function balanceOf(address _owner) public view returns (uint256) {
    return balances[_owner];
}

/**
 * Transfer Beercoins to another user
 *
 * @param _to the address of the recipient
 * @param _value the amount to send
 */
function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != 0x0);

    uint256 balanceFrom = balances[msg.sender];

    require(_value <= balanceFrom);

    uint256 oldBalanceTo = balances[_to];
    uint256 newBalanceTo = oldBalanceTo + _value;

    require(oldBalanceTo <= newBalanceTo);

    balances[msg.sender] = balanceFrom - _value;
    balances[_to] = newBalanceTo;

    Transfer(msg.sender, _to, _value);

    return true;
}

/**
 * Transfer Beercoins from other address if a respective allowance exists
 *
 * @param _from the address of the sender
 * @param _to the address of the recipient
 * @param _value the amount to send
 */
function transferFrom(address _from, address _to, uint256 _value) public returns (bool) {
    require(_to != 0x0);
```

```
        uint256 balanceFrom = balances[_from];
        uint256 allowanceFrom = allowances[_from][msg.sender];

        require(_value <= balanceFrom);
        require(_value <= allowanceFrom);

        uint256 oldBalanceTo = balances[_to];
        uint256 newBalanceTo = oldBalanceTo + _value;

        require(oldBalanceTo <= newBalanceTo);

        balances[_from] = balanceFrom - _value;
        balances[_to] = newBalanceTo;
        allowances[_from][msg.sender] = allowanceFrom - _value;

        Transfer(_from, _to, _value);

        return true;
    }

    /**
     * Allow another user to spend a certain amount of Beercoins on your behalf
     *
     * @param _spender the address of the user authorized to spend
     * @param _value the maximum amount that can be spent on your behalf
     */
    function approve(address _spender, uint256 _value) public returns (bool) {
        allowances[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        return true;
    }

    /**
     * The amount of Beercoins that can be spent by a user on behalf of another
     *
     * @param _owner the address of the user user whose Beercoins are spent
     * @param _spender the address of the user who executes the transaction
     */
    function allowance(address _owner, address _spender) public view returns (uint256) {
        return allowances[_owner][_spender];
    }
}

/**
 * A contract that defines a master with special debiting abilities
 * required for operating a user-friendly Beercoin redemption system
 */
contract MasteredBeercoin is ERC20Beercoin {
    address internal beercoinMaster;
```

```
mapping (address => bool) internal directDebitAllowances;

/**
 * Construct the MasteredBeercoin contract
 * and make the sender the master
 */
function MasteredBeercoin() public {
    beercoinMaster = msg.sender;
}

/**
 * Restrict to the master only
 */
modifier onlyMaster {
    require(msg.sender == beercoinMaster);
    _;
}

/**
 * The master of the Beercoin
 */
function master() public view returns (address) {
    return beercoinMaster;
}

/**
 * Declare a master at another address
 *
 * @param newMaster the new owner's address
 */
function declareNewMaster(address newMaster) public onlyMaster {
    beercoinMaster = newMaster;
}

/**
 * Allow the master to withdraw Beercoins from your
 * account so you don't have to send Beercoins yourself
 */
function allowDirectDebit() public {
    directDebitAllowances[msg.sender] = true;
}

/**
 * Forbid the master to withdraw Beercoins from you account
 */
function forbidDirectDebit() public {
    directDebitAllowances[msg.sender] = false;
}
```

```
/**
 * Check whether a user allows direct debits by the master
 *
 * @param user the user to check
 */
function directDebitAllowance(address user) public view returns (bool) {
    return directDebitAllowances[user];
}

/**
 * Withdraw Beercoins from multiple users
 *
 * Beercoins are only withdrawn this way if and only if
 * a user deliberately wants it to happen by initiating
 * a transaction on a plattform operated by the owner
 *
 * @param users the addresses of the users to take Beercoins from
 * @param values the respective amounts to take
 */
function debit(address[] users, uint256[] values) public onlyMaster returns (bool) {
    require(users.length == values.length);

    uint256 oldBalance = balances[msg.sender];
    uint256 newBalance = oldBalance;

    address currentUser;
    uint256 currentValue;
    uint256 currentBalance;
    for (uint256 i = 0; i < users.length; ++i) {
        currentUser = users[i];
        currentValue = values[i];
        currentBalance = balances[currentUser];

        require(directDebitAllowances[currentUser]);
        require(currentValue <= currentBalance);
        balances[currentUser] = currentBalance - currentValue;

        newBalance += currentValue;

        Transfer(currentUser, msg.sender, currentValue);
    }

    require(oldBalance <= newBalance);
    balances[msg.sender] = newBalance;

    return true;
}

/**
```

```
 * Withdraw Beercoins from multiple users
 *
 * Beercoins are only withdrawn this way if and only if
 * a user deliberately wants it to happen by initiating
 * a transaction on a plattform operated by the owner
 *
 * @param users the addresses of the users to take Beercoins from
 * @param value the amount to take from each user
 */
function debitEqually(address[] users, uint256 value) public onlyMaster returns (bool) {
    uint256 oldBalance = balances[msg.sender];
    uint256 newBalance = oldBalance + (users.length * value);

    require(oldBalance <= newBalance);
    balances[msg.sender] = newBalance;

    address currentUser;
    uint256 currentBalance;
    for (uint256 i = 0; i < users.length; ++i) {
        currentUser = users[i];
        currentBalance = balances[currentUser];

        require(directDebitAllowances[currentUser]);
        require(value <= currentBalance);
        balances[currentUser] = currentBalance - value;

        Transfer(currentUser, msg.sender, value);
    }

    return true;
}

/**
 * Send Beercoins to multiple users
 *
 * @param users the addresses of the users to send Beercoins to
 * @param values the respective amounts to send
 */
function credit(address[] users, uint256[] values) public onlyMaster returns (bool) {
    require(users.length == values.length);

    uint256 balance = balances[msg.sender];
    uint256 totalValue = 0;

    address currentUser;
    uint256 currentValue;
    uint256 currentOldBalance;
    uint256 currentNewBalance;
    for (uint256 i = 0; i < users.length; ++i) {
```

```
            currentUser = users[i];
            currentValue = values[i];
            currentOldBalance = balances[currentUser];
            currentNewBalance = currentOldBalance + currentValue;

            require(currentOldBalance <= currentNewBalance);
            balances[currentUser] = currentNewBalance;

            totalValue += currentValue;

            Transfer(msg.sender, currentUser, currentValue);
        }

        require(totalValue <= balance);
        balances[msg.sender] = balance - totalValue;

        return true;
    }

    /**
     * Send Beercoins to multiple users
     *
     * @param users the addresses of the users to send Beercoins to
     * @param value the amounts to send to each user
     */
    function creditEqually(address[] users, uint256 value) public onlyMaster returns (bool) {
        uint256 balance = balances[msg.sender];
        uint256 totalValue = users.length * value;

        require(totalValue <= balance);
        balances[msg.sender] = balance - totalValue;

        address currentUser;
        uint256 currentOldBalance;
        uint256 currentNewBalance;
        for (uint256 i = 0; i < users.length; ++i) {
            currentUser = users[i];
            currentOldBalance = balances[currentUser];
            currentNewBalance = currentOldBalance + value;

            require(currentOldBalance <= currentNewBalance);
            balances[currentUser] = currentNewBalance;

            Transfer(msg.sender, currentUser, value);
        }

        return true;
    }
}
```

```
/**
 * A contract that defines the central business logic
 * which also mirrors the life of a Beercoin
 */
contract Beercoin is MasteredBeercoin {
    event Produce(uint256 newCaps);
    event Scan(address[] users, uint256[] caps);
    event Burn(uint256 value);

    /**
     * Construct the Beercoin contract and
     * assign the initial supply to the creator
     */
    function Beercoin() public {
        balances[msg.sender] = INITIAL_SUPPLY;
    }

    /**
     * Increase the amounts of produced diamond, gold, silver, and
     * bronze bottle caps in respect to their occurrence probabilities
     *
     * This function is called if and only if a brewery has actually
     * ordered codes to produce the specified amount of bottle caps
     *
     * @param numberOfCaps the number of bottle caps to be produced
     */
    function produce(uint64 numberOfCaps) public onlyMaster returns (bool) {
        require(numberOfCaps <= producibleCaps);

        uint256 producedCaps = packedProducedCaps;

        uint64 targetTotalCaps = numberOfCaps;
        targetTotalCaps += uint64(producedCaps >> 192);
        targetTotalCaps += uint64(producedCaps >> 128);
        targetTotalCaps += uint64(producedCaps >> 64);
        targetTotalCaps += uint64(producedCaps);

        uint64 targetDiamondCaps = (targetTotalCaps - (targetTotalCaps % 10000)) / 10000;
        uint64 targetGoldCaps = ((targetTotalCaps - (targetTotalCaps % 1000)) / 1000) - targetDiamondCaps;
        uint64 targetSilverCaps = ((targetTotalCaps - (targetTotalCaps % 10)) / 10) - targetDiamondCaps - targetGoldCaps;
        uint64 targetBronzeCaps = targetTotalCaps - targetDiamondCaps - targetGoldCaps - targetSilverCaps;

        uint256 targetProducedCaps = 0;
        targetProducedCaps |= uint256(targetDiamondCaps) << 192;
        targetProducedCaps |= uint256(targetGoldCaps) << 128;
        targetProducedCaps |= uint256(targetSilverCaps) << 64;
        targetProducedCaps |= uint256(targetBronzeCaps);
```

```
    producibleCaps -= numberOfCaps;
    packedProducedCaps = targetProducedCaps;

    Produce(targetProducedCaps - producedCaps);

    return true;
}

/**
 * Approve scans of multiple users and grant Beercoins
 *
 * This function is called periodically to mass-transfer Beercoins to
 * multiple users if and only if each of them has scanned codes that
 * our server has never verified before for the same or another user
 *
 * @param users the addresses of the users who scanned valid codes
 * @param caps the amounts of caps the users have scanned as single 256-bit values
 */
function scan(address[] users, uint256[] caps) public onlyMaster returns (bool) {
    require(users.length == caps.length);

    uint256 scannedCaps = packedScannedCaps;

    uint256 currentCaps;
    uint256 capsValue;
    for (uint256 i = 0; i < users.length; ++i) {
        currentCaps = caps[i];

        capsValue = DIAMOND_VALUE * (currentCaps >> 192);
        capsValue += GOLD_VALUE * ((currentCaps >> 128) & 0xFFFFFFFFFFFFFFFF);
        capsValue += SILVER_VALUE * ((currentCaps >> 64) & 0xFFFFFFFFFFFFFFFF);
        capsValue += BRONZE_VALUE * (currentCaps & 0xFFFFFFFFFFFFFFFF);

        balances[users[i]] += capsValue;
        scannedCaps += currentCaps;
    }

    require(scannedCaps <= packedProducedCaps);
    packedScannedCaps = scannedCaps;

    Scan(users, caps);

    return true;
}

/**
 * Remove Beercoins from the system irreversibly
 *
 * @param value the amount of Beercoins to burn
```

```
     */
    function burn(uint256 value) public onlyMaster returns (bool) {
        uint256 balance = balances[msg.sender];
        require(value <= balance);

        balances[msg.sender] = balance - value;
        burntValue += value;

        Burn(value);

        return true;
    }
}
```