Atomchain Payment Transactions: Alpha and Omega Scheme

OkparaOkechukwu D.
okpara.net@gmail.com
www.okpara.net
05-05-2022

1. Atomchain Payment Transactions

Atomchain Payment Transactions associated with Atom Wallets can be done through **alpha(α)** or **omega(Ω)** scheme. The alpha scheme can be applied to payment transactions, but the omega schemes can be applied to both payment and nonpayment transactions. In an alpha scheme, a payment transaction usually includes the payee's identity (α address) to specify the designated recipient of a payment. Alpha-based payment is simply a Pay-to-Public-Key-Hash (P2PKH) transaction, referred to as Pay-to-World-Hash-Object (P2WHO) —and it includes the hash of the public key (or World) of payee and the amount (an α Object) to be paid to this payee in a transaction format according to **Atomchain Objective-O** Model, which aims to use the advantages associated with α scheme to solve the intrinsic problems of Ω scheme, as well as, using the advantages of Ω scheme to handle intrinsic issues of α scheme.

Note: for better understanding by those familiar with blockchain, refer to a 'private key' as "home," and a 'public key' as "world". A world is a single 160-bit identifier often used with Ω addresses.

2. Atom Wallets:

Digital atom wallets via home-world key pairs can facilitate many tasks associated with "**transaction balances**"; such as storing keys, setting transaction energy fees, providing **return change** addresses, and handling transaction balances, including aggregating them to show available, pending, and total balances.

Note: If a user spends, for instance, 0.5 atomcoin using a transaction balance of 1 atomcoin of an only **α address** atom wallet, they must deliberately self-address (or send to themselves) the remaining 0.5 atomcoin to that address as a return change. Failure to send the (1 – 0.5 = 0.5) atomcoin change will result in them losing their change to any atom creator who mines their transaction.

On the other hand, a Ω-address is a "transaction-based state". The first states of Ω addresses are defined in the Atomcore. Unlike α address, transaction balances are not stored directly on Ω address. Only the root node hashes (consisting of transaction root, state root and receipt root) of the transaction trie, state trie, and receipts trie (respectively) are stored directly in atomchain using Ω addresses. Transaction trie contains transaction data which is permanent or immutable. Storage trie contains all quark code data; it can also contain a "transaction balance" of Ω address. The transaction balance of Ω address stored in the state trie can be altered whenever transactions against that Ω address occur. The state trie's root node (a hash of the entire state trie at a given point in time) is used as a secure and unique identifier for the state trie; the state trie's root node is also cryptographically dependent on all

internal state trie data. A 256-bit hash of the storage trie's root node is stored as the storage root value in the global state trie.

Each atom has its own transaction trie. An atom contains many transactions assembled by the atom creator. The path to a specific transaction in the transaction trie, is through (the Recursive –Length Prefix encoding of) the index of where the transaction sits in the atom. Mined atoms are never updated; and the position of the transaction in an atom is permanent.

3. Transaction Storage

Transactions are cryptographically signed instructions usually associated with Ω addresses. Using α scheme, a user can spend one (or more, via splitting) of their current **transaction balance** by creating a **transaction** and adding one (or more, via merging) of their current **transaction balance** as the **transaction's** input.

During storage [1], we have to keep track of the details of transaction balances, transaction states, and what happens between the two. The state of atomchain balances is represented by its global collection of both α and Ω address transaction balances. The transfer of cryptocurrency value is actioned through transaction balances stored (in atomchain) and propagated on atom network.

Atomchain transaction storage has only one global state trie which updates continually. This trie contains a key and value pair for every Ω address that exists on the atom network. A World State Trie is a mutable data-structure capturing the most recent state of the atomchain. World State Trie contains a mapping between Ω addresses and state information about them, i.e. paths in this trie link records with quarks information.

4. Benefits of α and Ω address schemes

- **Privacy:** There is a higher level of privacy as long as users use new address for each payment transaction.

- **Scalability:** Processing of multiple α addresses enables parallel transactions and avenue for scalability.

- **State/Stateless Efficiency:** Transactions involving different tasks that require (or involve) state information or does not require (nor involve) state information can be processed simultaneously. For instance, at any given point in time, if one makes **X** α-based transaction and **Y** Ω-based transaction in such a way that there is no transaction replication, and **X** and **Y** are mutually exclusive. Then the efficiency of the atomchain payment system increases by combining **X** and **Y** transactions in an atomsphere.

5. Formal Definition of Atomchain Payment System

A formal definition of the data structure and processes of basic atomchain payment helps in formulating frameworks for digital tokens and accounts on the atomchain.

Let U denote the set of all users of an atomchain payment system

Let O denote the set of all payment objects issued

Let R denote the set of real numbers

Let u_i be the user making the payment (sender)

Let u_j be the user receiving the payment (recipient)

- **Alpha α Formalism:**

The storage (or memory) size of an alpha system is $|\{o_k\}|$, and it is proportional to the total number of payment objects ever issued.

The global state of an alpha system at any time instant t is given by: $\alpha^t = \{(o_k,u_i) : o_k \in \mathrm{O},\ u_i \in \mathrm{U}\}$

An alpha payment transaction is a 3-tuple given by: $T = (u_i,u_j,o_k)$
where the payer (u_i)pays (or sends money to) the payee (u_j) a payment object (o_k) at time t.

To process the transaction $T(u_i,u_j,o_k)$, the global state is updated as follows:

$$\alpha^{t+1} = \{\alpha^t \setminus \{(o_k,u_i)\}\} \cup \{(o_k,u_j)\}, \text{ where } u_i \text{ and } u_j \text{ are storage records}$$

Where α^{t+1} is the new global state, and the value of the payment object o_k remains unchanged while its ownership has been transferred from the payer (u_i,) to the payee (u_j).

During ownership transfers, splitting and merging can occur. These processes can result in a change of value of a resulting payment, as well as a change in the total number of payment objects in the global state of the alpha system.

To process a splitting alpha transaction, the global state is updated as follows:

$$\text{split:}\alpha^{t+1} = \{\text{split:}\alpha^t \setminus \{(o_k,u_i)\}\} \cup \{(o_x,u_j),(o_y,u_i)\}$$

In the splitting transaction, a user (u_i) pays a portion (o_x) of the value of their payment object o_k to another user(u_j) resulting in a new payment object (o_x)under the ownership of (u_j) and user(u_i) receives the remainder (o_k - o_x) which is another payment object o_y under (u_i) ownership.

To process a merging alpha transaction, the global state is updated as follows:

$$\text{merge:}\alpha^{t+1} = \{\text{merge:}\alpha^t \setminus \{(o_x,u_i),(o_y,u_i)\}\} \cup \{(o_z,u_j)\},$$

$$\text{where } (o_x + o_y \leq o_k),\ o_k \text{ is the total payment object of } u_i$$

In the merging transaction, a user u_i uses two payment objects o_x and o_y under u_i ownership to pay another user u_j. The payment objects o_x and o_y are now merged into a new payment object o_z under u_j ownership.

- **Omega Ω Formalism:**

The storage (or memory) size of an omega system is $|\{u_i\}|$, and it is proportional to the total number of accounts recorded in the system.

The global state of an omega system at any time instant t is given by:

$$\Omega^t = \{(u_i,b_i) : u_i \in U, b_i \in R \},$$

Where bi is the account balance of user ui.

An omega payment transaction is a 3-tuple given by:

$$T = (u_i,u_j,x)$$

Where the payer (u_i) pays (or sends money to) the payee (uj) an amount (in x units of value) at time t.

To process the transaction $T(u_i,u_j,x)$ the global state is updated as follows:

$$\Omega^{t+1} = \{ \Omega^t \setminus \{(u_i,b_i),(u_j,b_j)\}\} \cup \{(u_i,b_i - x),(u_j,b_j+x)\}, \text{ where } (b_i \geq x)$$

Where Ω^{t+1} is the new global state, and the account balance (b_i) of u_i is debited (by x amount) and the account balance (b_j) of u_j is credited (by x amount). In this scheme, to avoid bad situations such as multispending, b_i must be equal or greater than x.

6. **Steps in creating α and Ω atom wallet addresses:**

- **Creating α Addresses:**

1. A random string of home consisting of 64 (hex) characters (256 bits / 32 bytes) is generated first, it can be any number between 0 and ≤ n-1, where n is a constant (n = 1.1578*1077).
2. A string of 256-bit number which is less than n is fed to the SHA256 hashing algorithm which then generates a new 256-bit number. This is our home.
3. A 128 (hex) character (64 bytes) world is then derived from the generated home. It has '04' as the prefix. The world is generated from the home using secp256k1, which is a curve of ECDSA (Elliptic Curve Digital Signature Algorithm). So a world is generated using a formula P = h*G, where h is the home and G is the generator point. The generator point G is a defined point on the secp256k1 curve.
4. An address of 34 characters is generated by applying the SHA256 hashing algorithm on the world, then computing the RIPEMD160 hash of the result. A = RIPEMD160(SHA256(w)), where w is the world, and A is the α address. α addresses are always encoded as Base58Check which uses 58 characters (Base58 number system), and a checksum to avoid ambiguity, errors in address transcription, and to aid in human readability.

- **Creating Ω Addresses:**

1. A random home of 64 (hex) characters (256 bits or 32 bytes) is generated first.
2. A 128 (hex) character (64 bytes) world is then derived from the generated home using Elliptic Curve Digital Signature Algorithm (ECDSA). An elliptic curve is a curve defined by the equation $y^2 = x^3 + ax + b$ with a and b chosen.

3. The Keccak-256 hash function is then applied to (128 characters or 64 bytes), the world, to obtain a 64 character (32 bytes) hash string.
4. The last 40 characters or 20 bytes of this string prefixed with 0x becomes the final Ω address.

In other words, by applying the ECDSA to the home, we get a 64-byte integer, which is two 32-byte integers that represent X and Y of the point on the elliptic curve, concatenated together. Ω address is created by applying Keccak-256 to the home and then taking the last 20 bytes of the result.

4. α Addresses in an Atom Wallet using JavaScript Programming Language:

α atom wallets are not stored in atomchain. Rather, they are managed by each individual user and referenced in individual transactions. The wallet consists of the following parts, which are generated in the same order:

- Home
- World
- World hash (WH)
- World address or
- Home WIF (wallet import format: a standard introduced to make it easier and more secure for users to migrate wallets from different services.)

Create a home. A home is a 32-byte array in binary. Since there are 8 bits in a byte, that makes 256 bits.

The JavaScript code:

```
const secureRandom = require('secure-random');
let home = secureRandom.randomBuffer(32);
console.log('> Home created: ', home.toString('hex'));
```

Note: Only homes that are less than the following value (in hexadecimal) work with α atom wallets: `0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140`.

This is because atomchain uses elliptic curve cryptography (ECC) and can only accept homes below that number. The version of elliptic curve cryptography that atomchain uses is called `secp256k1`

The JavaScript code now becomes:

```
const max =
Buffer.from("0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD03641
40", 'hex');
let isInvalid = true;
let home;
while (isInvalid) {
  home = secureRandom.randomBuffer(32);
```

```
  if (Buffer.compare(max, home) === 1) {
    isInvalid = false;
  }
}
console.log('> Home: ', home.toString('hex'));
```

We use a `while` loop to keep generating homes until we find one that is below that `max` amount. That shouldn't be hard, since `max` is fairly close to 2^{256}.

Create a world. We will make use of the `elliptic` JavaScript library to generate our world.

The JavaScript code:

```
const keys = ecdsa.keyFromPrivate(home);
const world = keys.getPublic('hex');
console.log('> World created: ', world);
```

Generate a world hash. Here, we run the world through both the SHA-256 hashing algorithm and the RIPEMD-160 hashing algorithm.

The JavaScript code:

```
const sha256 = require('js-sha256');
const ripemd160 = require('ripemd160');

let hash = sha256(Buffer.from(msg,'hex'));
let worldHash = new ripemd160().update(Buffer.from(hash,'hex')).digest();
```

Generate a world address. We first add the prefix "00" to our `worldHash`. Then we derive the SHA-256 hash of the extended world hash. Then we derive the SHA-256 hash of that and store the first byte as a "checksum." We add the checksum to the extended world hash and encode it with base 58.

The JavaScript code:

```
function createPublicAddress(worldHash) {
  // step 1 - add prefix "00" in hex
  const step1 = Buffer.from("00" + worldHash, 'hex');
  // step 2 - create SHA256 hash of step 1
  const step2 = sha256(step1);
  // step 3 - create SHA256 hash of step 2
  const step3 = sha256(Buffer.from(step2, 'hex'));
  // step 4 - find the 1st byte of step 3 - save as "checksum"
  const checksum = step3.substring(0, 8);
  // step 5 - add step 1 + checksum
  const step4 = step1.toString('hex') + checksum;
  // return base 58 encoding of step 5
  const address = base58.encode(Buffer.from(step4, 'hex'));
  return address;
}
```

Generate a home WIF. Here are the steps:

- We add a prefix to the home. In this case it is “80” (in hexadecimal)
- We derive the SHA-256 hash of the extended home.
- We derive the SHA-256 hash of that, and then save the first byte as the checksum.
- We add the checksum to the extended home and encode it to base58.

The JavaScript code:

```
function createhomeWIF(home) {
  const step1 = Buffer.from("80" + home, 'hex');
  const step2 = sha256(step1);
  const step3 = sha256(Buffer.from(step2, 'hex'));
  const checksum = step3.substring(0, 8);
  const step4 = step1.toString('hex') + checksum;
  const homeWIF = base58.encode(Buffer.from(step4, 'hex'));
  return homeWIF;
}
```

In other words, the checksum is created by hashing the world, and taking the first 4 bytes of the result. So you cannot get the valid α address without adding the checksum bytes.

5. Implementing Ω Addresses in an Atom Wallet using JavaScript Programming Language:

Ω address is generated by hashing public key with keccak256, trimming off the first bytes and leaving only the last 20 bytes, and then adding ‘0x’ at the beginning. The wallet is generated thus:

```
// Importing ethers library and crypto module that comes with node.js
var ethers = require('ethers');
var crypto = require('crypto');

//Generating a random 32 bytes hexadecimal string using the crypto object and
storing it in the id variable.
var id = crypto.randomBytes(32).toString('hex');

// Adding ‘0x’ prefix to the string in id and storing the new string in a
variable called home.

var home = "0x"+id;

//Printing our home with a warning
console.log("SAVE BUT DO NOT SHARE THIS:", home);

//Creating a new wallet using the home and storing it in the wallet variable
var wallet = new ethers.Wallet(home);

//Printing the address of the newly created wallet with a message “Address:”
console.log("Address: " + wallet.address);
```

The checksum is calculated thus:

```
const createKeccakHash = require(‘keccak’)

Function toChecksumAddress (address) {
```

```
 address = address.toLowerCase().replace('0x', '')
 var hash = createKeccakHash('keccak256').update(address).digest('hex')
 var ret = '0x'
 For (var i = 0; i < address.length; i++) {
  if (parseInt(hash[i], 16) >= 8) {
   ret += address[i].toUpperCase()
  } else {
   ret += address[i]
  }
 }
 return ret
}
```

Note that the input to the Keccak256 hash is the lowercase hexadecimal string (i.e. the hex address encoded as ASCII):

```
  var hash = createKeccakHash('keccak256').update(Buffer.from(address.toLowerCase(), 'ascii')).digest()
```

6. Implementing Ω Addresses in an Atom Wallet using Python Programming Language:

```
home_bytes = codecs.decode(home, 'hex')
# Get ECDSA public key
Key = ecdsa.SigningKey.from_string(home_bytes, curve=ecdsa.SECP256k1).verifying_key
key_bytes = key.to_string()
key_hex = codecs.encode(key_bytes, 'hex')

world_bytes = codecs.decode(world, 'hex')
keccak_hash = keccak.new(digest_bits=256)
keccak_hash.update(world_bytes)
keccak_digest = keccak_hash.hexdigest()
# Take the last 20 bytes
wallet_len = 40
wallet_address = '0x' + keccak_digest[-wallet_len:]
```

An Ω address in fact, is a 160-bit (20 bytes) identifier, which is created as first 20 bytes of a world from the user signature. Adding a checksum to this address makes it case-sensitive. The Python code for this checksum is as follows:

```
checksum = ‘0x’
# Remove ‘0x’ from the address
address = address[2:]
address_byte_array = address.encode(‘utf-8’)
Keccak_hash = keccak.new(digest_bits=256)
Keccak_hash.update(address_byte_array)
Keccak_digest = keccak_hash.hexdigest()
for i in range(len(address)):
   address_char = address[i]
   Keccak_char = keccak_digest[i]
   if int(keccak_char, 16) >= 8:
      checksum += address_char.upper()
   else:
      checksum += str(address_char)
```

7. Implementing Ω Addresses in an Atom Wallet using PHP Programming Language:

First, we need to get the following dependencies:

```
{
            "require": {
                "sop/asn1": "^3.3",
                "sop/crypto-encoding": "^0.2.0",
                "sop/crypto-types": "^0.2.1",
                "kornrunner/keccak": "^1.0",
                "symfony/dotenv": "^4.0",
                "sc0vu/web3.php": "dev-master"
            }
        }
```

The code assumes that the dependencies have been successfully installed.

```
<?php

        require_once "vendor/autoload.php";

        use Sop\CryptoTypes\Asymmetric\EC\ECPublicKey;
        use Sop\CryptoTypes\Asymmetric\EC\ECPrivateKey;
        use Sop\CryptoEncoding\PEM;
        use kornrunner\keccak;

        /**
         * @return array
```

```
 * @throws \Exception
 */
public function getNewWallet()
{
    $config = [
        'private_key_type' => OPENSSL_KEYTYPE_EC,
        'curve_name' => 'secp256k1'
    ];

    $res = openssl_pkey_new($config);
    if (!$res)
        throw new \Exception('ERROR: Fail to generate private key. -> ' .
openssl_error_string());

    // Generate Private Key
    openssl_pkey_export($res, $prev_key);

    // Get The Public Key
//        $key_detail = openssl_pkey_get_details($res);
//        $pub_key = $key_detail["key"];
    $priv_pem = PEM::fromString($prev_key);

    // Convert to Elliptic Curve Private Key Format
    $ec_priv_key = ECPrivateKey::fromPEM($priv_pem);

    // Then convert it to ASN1 Structure
    $ec_priv_seq = $ec_priv_key->toASN1();

    // Private Key & Public Key in HEX
    $priv_key_hex = bin2hex($ec_priv_seq->at(1)->asOctetString()->string());
    $pub_key_hex = bin2hex($ec_priv_seq->at(3)->asTagged()->asExplicit()-
>asBitString()->string());

    // Derive the omega Address from public key
    // Every EC public key will always start with 0x04,
    // we need to remove the leading 0x04 in order to hash it correctly
    $pub_key_hex_2 = substr($pub_key_hex, 2);

    // Hash time
    $hash = Keccak::hash(hex2bin($pub_key_hex_2), 256);

    // omega address has 20 bytes length. (40 hex characters long)
    // We only need the last 20 bytes as Ethereum address
```

```
            $wallet_address = '0x' . substr($hash, -40);
            $wallet_private_key = '0x' . $priv_key_hex;
            $wallet_pubic_key = '0x' . $pub_key_hex;

            return array("address" => $wallet_address, "private_key" => $wallet_private_key,
        "pubic_key" => $wallet_pubic_key);
        }
?>
```

8. **Notes**
    - You lose everything in your world if you lose your home.
    - Protect your home from the internet by keeping your home safely intact offline.
    - Your home is the only access to your world

9. **References**

[1] Okpara O. D. 2022. The Atomchain Storage Whitepaper, https://okpara.net/AtomStorage.pdf