Orc Protocol Specification

Version 2.2 (August 15, 2017)

Gordon Hall (gordonh@member.fsf.org)
Ryan Foran
Braydon Fuller

0 License

1 Introduction

This specification documents the Orc network protocol in its entirety for the purpose of enabling its implementation in other languages. Described here, is the protocol **base** - the minimum specification for compatibility with the Orc network. Additional optional extensions to this work are defined as Orc Improvement Proposals (or "IMPs").

2 Identities

Every node (host computer speaking the Orc protocol) on the network possesses a unique cryptographic identity. This identity is used to derive a special 160 bit identifier for the purpose of organizaing the overlay structure and routing messages *(3.1: Kademlia)*. In order for a node to join the network it must generate an identity.

Identities are described as **hierarchically deterministic** and serve the purpose of running a cluster of nodes that can all share the same contracts and act on behalf of each other in the network. The specification extends Bitcoin ECDSA derivation standard BIP32 and BIP43.

Key derivation must match the specification of Bitcoin Hierarchical Deterministic Wallets (BIP32) with the purpose field described in Bitcoin Purpose Field for Deterministic Wallets (BIP43).

We define the following levels in BIP32 path:

```
m / purpose' / group_index' / node_index
```

The apostrophe in the path indicates that BIP32 hardened derivation is used. Purpose is a constant set to 3000, so as to not collide with any bitcoin related proposals which recommends to use the BIP number.

```
m / 3000' / group_index' / node_index
```

The `group_index` for most purposes will be 0. However is reserved for a future use to be able to increment in the case that the contracts should be updated with a new key. The `node_index` can be a number from 0 through 2 ^ 31 - 1, so that it's using a non-hardened paths and it's always possible to derive the public key for a node using the `m / 3000' / group_index'` derived extended public key. This gives a total of 2.147 billion possible nodes to run in a group cluster.

> As noted in BIP32, a compromised private key at the `node_index` level in combination with the extended public key at the `group_index` level will compromise all descending private keys derived from the `group_index` level, this is the rationale for a hardened path for the `group_index`.

In every message exchanged on the network, each party will include a tuple structure which includes enough information to locate and authenticate each party.

```
["<node_id>", { /* <contact> */ }]
```

2.1 Node ID Generation

Once a HD identity has been generated, a child identity should be derived and used for a single node. The resulting public key from that child identity is used to derive the Node ID. The node's identifier is the `RIPEMD160( SHA256( CHILD_PUBLIC_KEY ) )` encoded in hexidecimal. This value is inserted as the first item in the identity tuple.

```
["705e93f855e60847fda4c48adff0dc1b1f7c40ef", { /* <contact> */ }]
```

2.2 Contact Hash Map

The second entry in the identity tuple contains additional information specific to addressing the node on the network. This includes:

```
{
  "hostname": "ip.address.or.domain.name",
  "port": 8443,
  "protocol": "https:",
  "xpub": "<child_identity_public_extended_key>",
  "index": "<child_identity_derivation_index>"
}
```

Additional properties may be included based on individual use cases within the network, however the properties above are **required**.

3 Network Structure

Orc employs a **structured** network, meaning that nodes are organized and route messages based on a deterministic metric. The network uses a Kademlia distributed hash table as the basis for the network overlay. In addition to Kademlia, Orc also employs other extensions to mitigate issues and attacks defined by the work on S/Kademlia.

In addition to the distributed hash table, Orc also implements a publish-subscribe system, Quasar, atop the Kademlia overlay to provide effective delivery of publications related to the solicitation of storage space *(6: Storage Contracts)*.

3.1 Kademlia

Once an Orc node has completed generating its identity, it bootstraps its routing table by following the Kademlia "join" procedure. This involves querying a single known "seed" node for contact information about other nodes that possess a Node ID that is close (XOR distance) to its own *(4.4 FIND_NODE)*. This is done iteratively, sending the same query to the `ALPHA` (3) results that are closest, until the further queries no longer yield results that are closer or the routing table is sufficiently bootstrapped.

3.2 Quasar

Upon successfully bootstrapping a routing table, a node may choose to subscribe to certain publication topics related to types of storage contracts they wish to accept *(6.2 Topic Codes)*. Each node in the network, maintains an attenuated bloom filter, meaning a **list** of exactly 3 bloom filters, each containing the the topics in which neighboring nodes are interested.

```
Filter 0 [...] - WE are subscribed
Filter 1 [...] - 3 NEAREST NEIGHBORS are subscribed
Filter 2 [...] - NEIGHBORS' 3 NEAREST NEIGHBORS' are subscribed
```

The Orc network expects these blooms filters to be constructed and modified in a specific manner. Each filter's bitfield must be exactly 160 bits in size. Items are hashed with FNV in a manner consistent with the paper "Less Hashing, Same Performance: Building a Better Bloom Filter" using 2 slices. To illustrate the hashing in pseudo-code:

```
function calc(key, size = 160, slices = 2) {
  function fnv(seed, data) {
    const hash = new FNV();

    hash.update(seed);
    hash.update(data);

    return hash.value() >>> 0;
  }

  const hash1 = fnv(Buffer.from([83]), key);
  const hash2 = fnv(Buffer.from([87]), key);
  const hashes = [];

  for (let i = 0; i < slices; i++) {
    hashes.push((hash1 + i * hash2) % size);
  }

  return hashes;
}
```

The above example illustrates how to calculate the values to insert into a bloom filter for a given key (or topic code). To subscribe to a given topic, the code(s) should be processed as shown above and then inserted into the filter at index 0. Once the filter at index 0 represents what the node wants to receive, it must exchange this information with its 3 nearest neighbors *(4.7 SUBSCRIBE + 4.8 UPDATE)*. This allows publications to be properly relayed to nodes who are most likely to be subscribed to the given topic.

3.3 Transport

The Orc network operates entirely over HTTPS. TLS *must* be used - there is no cleartext supported. In general this means that certificates are self-signed and you must accept them in order to communicate with others on the network. In addition, Orc operates exclusively over Tor and because of this, there is no need to perform certificate pinning for SSL since all Orc nodes are Tor hidden services and authentication is performed by the nature of Tor's routing.

Each Orc node exposes endpoints to other nodes; one for receiving RPC messages *(4. Remote Procedure Calls)*, one for serving and accepting raw data streams associated with held contracts *(5. Data Transfer Endpoints)*, and one for delivering identity metadata to requesters *(3.4 Root Endpoint)*.

Requests sent to the RPC endpoint require a special HTTP header `x-kad-message-id` to be included that matches the `id` parameter in the associated RPC message *(4.1 Structure and Authentication)*.

3.4 Root Endpoint

Whenver an Orc node receives a GET request at the root path (/), it must with it's contact metadata:

```
["705e93f855e60847fda4c48adff0dc1b1f7c40ef", { /* <contact> */ }]
```

This is to enable new Orc nodes who have yet to discover this information to query seeds for it before bootstrapping their routing table.

4 Remote Procedure Calls

- **Method:** `POST`
- **Path:** `/rpc/`
- **Content Type:** `application/json`
- **Headers:** `x-kad-message-id`

4.1 Structure and Authentication

Each remote procedure call sent and received between nodes is composed in the same structure. Messages are formatted as a JSON-RPC 2.0 *batch* payload containing 3 objects. These objects are positional, so ordering matters. The anatomy of a message takes the form of:

```
[{ /* rpc */ },{ /* notification */ },{ /* notification */ }]
```

At position 0 is the RPC request/response object, which must follow the JSON-RPC specification for such an object. It must contain the properties: `jsonrpc`, `id`, `method`, and `params` if it is a request. It must contain the properties: `jsonrpc`, `id`, and one of `result` or `error` if it is a response.

At positions 1 and 2 are a JSON-RPC notification object, meaning that it is not required to contain an `id` property since no response is required. These two notifications always assert methods `IDENTIFY` and `AUTHENTICATE` respectively. Together, these objects provide the recipient with information regarding the identity and addressing information of the sender as well as a cryptographic signature to authenticate the payload.

> Positions 3 and beyond in this structure are reserved for future protocol extensions related to global message processing.

Example: Request

```
[
  {
    "jsonrpc": "2.0",
    "id": "<uuid_version_4>",
    "method": "<method_name>",
    "params": ["<parameter_one>", "<parameter_two>"]
  },
  {
    "jsonrpc": "2.0",
    "method": "IDENTIFY",
    "params": [
      "<public_key_hash>",
      {
        "hostname": "sender.hostname",
        "port": 8443,
        "protocol": "https:",
        "xpub": "<public_extended_key>",
        "index": "<child_key_derivation_index>"
      }
    ]
  },
  {
    "jsonrpc": "2.0",
    "method": "AUTHENTICATE",
    "params": [
      "<payload_signature>",
      "<child_public_key>",
      ["<public_extended_key>", "<child_key_derivation_index>"]
    ]
  }
]
```

Example: Response

```
[
  {
    "jsonrpc": "2.0",
    "id": "<uuid_version_4_from_request>",
    "result": ["<result_one>", "<result_two>"]
  },
  {
    "jsonrpc": "2.0",
    "method": "IDENTIFY",
    "params": [
      "<public_key_hash>",
      {
        "hostname": "receiver.hostname",
        "port": 8443,
        "protocol": "https:",
        "xpub": "<public_extended_key>",
        "index": "<child_key_derivation_index>"
      }
    ]
  },
  {
    "jsonrpc": "2.0",
    "method": "AUTHENTICATE",
    "params": [
      "<payload_signature>",
      "<child_public_key>",
      ["<public_extended_key>", "<child_key_derivation_index>"]
    ]
  }
]
```

In the examples above, `public_key_hash` and `child_public_key` must be encoded as hexidecimal strings, `public_extended_key` must be encoded as a base58 string (in accordance with BIP32), and `payload_signature` must be encoded as a base64 string which is the concatenation of the public key recovery number with the actual signature of the payload - excluding the object at index 2 (`AUTHENTICATE`). This means that the message to be signed is `[rpc, identify]`.

> Note the exclusion of a timestamp or incrementing nonce in the payload means that a man-in-the-middle could carry out a replay attack. To combat this, it is urged that the `id` parameter of the RPC message (which is a universally unique identifier) be stored for a reasonable period of time and nodes should reject messages that attempt to use a duplicate UUID.

The rest of this section describes each individual method in the base protocol and defines the parameter and result signatures that are expected. If any RPC message yields an error, then an `error` property including `code` and `message` should be send in place of the `result` property.

4.2 `PING`

This RPC involves one node sending a `PING` message to another, which presumably replies. This has a two-fold effect: the recipient of the `PING` must update the bucket corresponding to the sender; and, if there is a reply, the sender must update the bucket appropriate to the recipient.

Parameters: `[]`
Results: `[]`

4.3 `FIND_NODE`

Basic kademlia lookup operation that builds a set of K contacts closest to the the given key. The `FIND_NODE` RPC includes a 160-bit key. The recipient of the RPC returns up to K contacts that it knows to be closest to the key. The recipient must return K contacts if at all possible. It may only return fewer than K if it is returning all of the contacts that it has knowledge of.

Parameters: `[key_160_hex]`
Results: `[contact_0, contact_1, ...contactN]`

4.4 `FIND_VALUE`

Kademlia search operation that is conducted as a node lookup and builds a list of K closest contacts. If at any time during the lookup the value is returned, the search is abandoned. If no value is found, the K closest contacts are returned. Upon success, we must store the value at the nearest node seen during the search that did not return the value.

A `FIND_VALUE` RPC includes a B=160-bit key. If a corresponding value is present on the recipient, the associated data is returned. Otherwise the RPC is equivalent to a `FIND_NODE` and a set of K contacts is returned.

If a value is returned, it must be in the form of an object with properties: `timestamp` as a UNIX timestamp in milliseconds, `publisher` as a 160 bit public key hash in hexidecimal of the original publisher, and `value` which may be of mixed type that is valid JSON.

Parameters: `[key_160_hex]`
Results: `{ timestamp, publisher, value }` or `[...contactN]`

4.5 `STORE`

The sender of the `STORE` RPC provides a key and a block of data and requires that the recipient store the data and make it available for later retrieval by that key .

Parameters: `[key_160_hex, { timestamp, publisher, value }]`
Results: `[key_160_hex, { timestamp, publisher, value }]`

4.6 `SUBSCRIBE`

Upon receipt of a `SUBSCRIBE` message, we simply respond with a serialized version of our attenuated bloom filter. Senders of this message must merge the response with their local attenuated bloom filter starting at their filter at index 1.

Parameters: `[]`
Results: `[filter_0_hex, filter_1_hex, filter_2_hex]`

4.7 UPDATE

Upon receipt of an `UPDATE` message we merge the delivered attenuated bloom filter with our own. This is the inverse of `SUBSCRIBE`, where a peer requests a copy of our attenuated bloom filter.

Parameters: `[filter_0_hex, filter_1_hex, filter_2_hex]`
Results: `[]`

4.8 PUBLISH

Upon receipt of a `PUBLISH` message, we validate it, then check if we or our neighbors are subscribed. If we are subscribed, we execute our handler. If our neighbors are subscribed, we relay the publication to ALPHA random of the closest K. If our neighbors are not subscribed, we relay the publication to a random contact.

The parameters for a `PUBLISH` message are named, not positional. It must be a JSON object containing: `uuid`, a version 4 UUID string, `topic`, the topic string to which a node may be subscribed, `publishers`, an array of 160 bit public key hash strings in hexidecimal representing nodes that have relayed the message previously, `ttl` the number of hops left for relaying the publication, and `contents`, any arbitrary valid JSON data associated with the publication.

> Given the relatively high cost of a single publication's propagation through the overlay, nodes should take care to implement some reasonable rate limiting for relay of publications. It is advisable for nodes to ignore the `ttl` and refuse to relay a publication if the identifier at `publishers[0]` is matched as the originator more than 100 times within a minute.

Before relaying the message to others, we must add our public key hash to the `publishers` list and decrement the `ttl`.

Parameters: `{ uuid, topic, publishers, ttl, contents }`
Results: `[]`

4.9 CLAIM

Upon receipt of an `CLAIM` message, nodes must validate the descriptor, then ensure that there is enough available space for the shard. If both checks succeed, then the descriptor is signed and returned along with a consignment token so the initiating renter can immediately upload the data. This call is the functional inverse of `OFFER`, as it is used for a renter to signal to a farmer that it wishes to rent capacity. These messages are generally sent based on information collected when subscribed to farmer capacity publications.

Parameters: `[descriptor_map]`
Results: `[descriptor_map, token_256_hex]`

4.10 `CONSIGN`

Upon receipt of a `CONSIGN` message, the node must verify that it has a valid storage allocation and contract for the supplied hash and identity of the originator. If so, it must generate an authorization token which will be checked by the shard server before accepting the transfer of the associated shard.

Parameters: `[hash_160_hex]`
Results: `[token_256_hex]`

4.11 `AUDIT`

Upon receipt of an `AUDIT` message, the node must look up the contract that is associated with each hash-challenge pair in the payload, prepend the challenge to the shard data, and caclulate the resulting hash, formatted as a compact proof *(7 Retrievability Proofs)*.

Parameters: `[...{ hash, challenge }]`
Results: `[...{ hash, proof }]`

4.12 `MIRROR`

Upon receipt of a `MIRROR` message, the node must verify that it is in possesion of the shard on behalf of the identity or the message originator. If so, given the token-hash pair, it must attempt to upload it's copy of the shard to the target to establish a mirror. The originator must have an established contract with the target node and have issued a `CONSIGN` message to the target in advance to provide the `MIRROR` recipient with this token.

In addition to the hash and token, the sender must also include the target contact data in the form of `[public_key_hash, { hostname, port, xpub, index, protocol }]`.

Parameters: `[hash_160_hex, token_256_hex, target_contact]`
Results: `[status_message_string]`

4.13 `RETRIEVE`

Upon receipt of a `RETRIEVE` message, the node must verify that it is in possession of the shard on behalf of the identity of the originator. If so, it must generate an authorization token which will be checked by the shard server before accepting the transfer of the associated shard.

Parameters: `[hash_160_hex]`
Results: `[token_256_hex]`

4.14 `RENEW`

Upon receipt of a `RENEW` message, the recipient farmer must extend or terminate it's contract based on the new terms supplied by the renter. If the renewal descriptor is valid and complete, the farmer must store the updated version after signing and respond back to the originator with the version containing the updated signature *(6.1 Descriptor Schema)*.

Implementations should only allow certain properties to be updated: `ownerIdentity`, `ownerParentKey`, `ownerSignature`, `accessPolicies`, and `auditLeaves`. If the sender has attempted to modify any other parts of the contract, an error should be returned.

Parameters: `[descriptor_map]`
Results: `[descriptor_map]`

5 Data Transfer Endpoints

Initiating the transfer of data between nodes after a contract has been signed is straightforward. First, the initiator must request a transfer token from the custodian. If uploading the shard for the first time to a farmer, a CONSIGN RPC *(4.13 CONSIGN)* must be sent. If downloading the shard, a RETRIEVE RPC *(4.14 RETRIEVE)* is sent. The result of either of those messages should yield an authorization token that is included in the query string of the next request.

5.1 Uploading

- **Method:** `POST`
- **Path:** `/shards/{hash}?token={consign_token}`
- **Content Type:** `binary/octet-stream`

5.2 Downloading

- **Method:** `GET`
- **Path:** `/shards/{hash}?token={retrieve_token}`
- **Content Type:** `binary/octet-stream`

6 Storage Contracts

Nodes on the Orc network form agreements with each other regarding the storage of data by exchanging a "contract descriptor". These descriptors are included in contract publications intended to solicit storage space *(7 Renting Space)* as well as in `RENEW` messages *(4.15 RENEW)*. Contracts describe identifying attributes of the data, the parties involved, terms of payment, and information about audits *(8 Audit Preparation)*.

6.1 Contract Descriptor Schema

Contracts are structured as a flat JSON dictionary and includes cryptographic signatures from the parties involved. The properties that compose a contract are show as follows.

- `version` *Integer* Unique identifier for the contract (this is version 1)
- `ownerParentKey` *String* Node extended public key in base58
- `ownerIndex` *Integer* Derivation index for signature
- `ownerIdentity` *String* Node ID of the renter
- `ownerSignature` *String* Renter's cryptographic signature
- `providerParentKey` *String* Node extended public key in base58
- `providerIndex` *Integer* Derivation index for signature
- `providerIdentity` *String* Node ID of the farmer
- `providerSignature` *String* Farmer's cryptographic signature
- `shardSize` *Integer* Number of bytes to store
- `shardHash` *String* RIPEMD-160 SHA-256 hash of the data
- `auditLeaves` *Array* Merkle leaves for audit tree as hexidecimal strings
- `fundingDestination` *String* Address to send funds (optional)

For a completed and valid contract, all fields be assigned a value and both the renter and farmer must sign the contract, placing their signatures in the respective fields. The signature is generated by using the private key associated with the `renter_id` or `farmer_id`, which is the hash of the public key. The signature field must be the result of the public key recovery number prepended to the actual signature of the JSON formatted descriptor exluding both signature fields and all other fields sorted alphanumerically. The signature must be encoded as a base64 string.

6.2 Renting Space

When a node wishes to solicit storage capacity for a shard, it must know which nodes are both accepting contracts and have the available space. Short lived nodes renting data, like nodes who join for the sole purpose of storing some data then leave, will likely not stick around long enough to track enough capactity announcements *(6.4 Announcing Capacity)* to build a local cache of known farmers with space. For these nodes, they may fetch the latest known capacity data from a directory node.

6.3 Announcing Capacity

When a node joins the network and wishes to make some storage capacity available, it can proactively inform the network that it has available space. To announce capacity, a node must send a `PUBLISH` message to it's 3 nearest neighbors using topic `ANNOUNCE`.

The `contents` property should contain the number of bytes available for claim along with the farmer's contact information so that renters who wish to claim space are not required to perform a network walk to find the originator if it is not already in it's routing table. The value of this property should be:

```
[ { allocated, available }, [ identity, { contact } ] ]
```

Long lived renter nodes, like those operated by a directory on the network, may subscribe to these capacity announcements and maintain a backlog cache of providers who have space available. This allows these renting nodes to directly claim space and transfer data to the provider *(4.11 CLAIM)*.

Providers who announce capacity should do so on a reasonable interval to ensure that renters tracking their state have fresh information. It is reccommended to publish capacity announcements once per hour and no more than once per 15 minutes. Nodes should take care to refuse to relay publications originating from an identity who has published a capacity announcement more than 100 times within the last hour.

7 Retrievability Proofs

When the custodian of a data shard receives an audit *(4.12 AUDIT)*, it is expected to respond with proof that it still in possession of the shard. This works by computing a retrievability proof structure from a provided challenge. Upon receipt of a hash-challenge pair, the associated data is prepended with the challenge bytes and hashed:

```
RIPEMD160( SHA256( CHALLENGE + DATA ) )
```

The result of this operation yields a value that, when hashed again, equals one of the bottom leaves of the audit tree *(8 Audit Preparation)*. In addition to supplying this single-hashed value as proof that the farmer is still honoring the terms of the contract, the farmer must also provide the uncles required to rebuild the merkle tree. This proof response is specified as a series of nested JSON arrays.

```
[[[[["<response>"],"<uncle>"],"<uncle>"],"<uncle>"],"<uncle>"]
```

For clarification, given a simple merkle tree:

```
+-- HASH_0 (Root)
|   +-- HASH_1
|   |   +-- HASH_3
|   |   +-- HASH_4
|   +-- HASH_2
|   |   +-- HASH_5
|   |   +-- HASH_6 = ( RIPEMD160( SHA256( CHALLENGE_RESPONSE ) ) )
```

The resulting format of a proof for an audit matching `HASH_6` would appear as:

```
[HASH_1, [HASH_5, [CHALLENGE_RESPONSE]]]
```

The resulting format of a proof for an audit matching `HASH_3` would appear as:

```
[[[CHALLENGE_RESPONSE], HASH_5], HASH_2]
```

Upon receipt of the farmer's proof, the renter must verify that the proof is valid by using it to rebuild the merkle tree. If the proof is verified successfully, the renter is expected to issue a payment to the `payment_destination` defined in the original contract. The amount of the payment should be equal to: `payment_storage_price / audit_count` in addition to `payment_download_price * downloads_since_last_audit`.

If the verification fails then the contract is null and no payment is required. Conversely, if the verification succeeds and the renter does not issue the payment in a timely manner, then the contract is also null and the farmer may decide to cease storage of the data.

8 Audit Preparation

Before a renter node claims a capacity allocation, it must pre-calculate a series of "challenges". A challenge is simply 32 random bytes encoded as hexidecimal. The generated challenges must not be shared until the renter wishes to issue an `AUDIT` RPC for proof-of-retrievability.

An `auditLeaves` included in a contract contains the bottom leaves of a Merkle Tree. Each of the bottom leaves of the tree must be equal to the double `RIPEMD160( SHA256 ( challenge + shard ) )` encoded as hexidecimal. In order to ensure that the resulting merkle tree is properly balanced, the number of bottom leaves must be equal to the next power of 2 of the audit count. To ensure this, the additional leaves can simply be the double `RIPEMD160( SHA256 ( NULL ) )` (the same hash function for an audit, but applied to an empty buffer).

To audit a farmer is to request proof that it is still honoring the terms of the storage contract without the need to have it supply the entire shard of data. To do this, the renter must supply the farmer with one of the secret pre-calculated challenges *(4.12 AUDIT)*. The receiving farmer must respond with a retrievability proof *(7 Retrievability Proofs)*.

9 References

- Orc Improvement Proposals (`https://github.com/orcproject/imps`)
- BIP32 (`https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`)
- BIP43 (`https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki`)
- Kademlia (`http://www.scs.stanford.edu/~dm/home/papers/kpos.pdf`)
- S/Kademlia (`http://www.tm.uka.de/doc/SKademlia_2007.pdf`)
- Quasar (`https://www.microsoft.com/en-us/research/wp-content/uploads/2008/02/iptps08-quas`
- FNV (`https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function`)
- Less Hashing, Same Performance: Building a Better Bloom Filter (`http://www.eecs.harvard.edu/~michaelm/postscripts/rsa2008.pdf`)
- Merkle Tree (`https://en.wikipedia.org/wiki/Merkle_tree`)