🧠 Outpost Propagation Protocol

⚠️ THIS NO LONGER IS ACCURATE; OPP implementation will use protobufs. A refactor of ethereum is underway

Introduction

> *You down with O.P.P.?*
> *(Yeah you know me)*
>
> – Naughty by Nature

OPP defines a mechanism for synchronizing between an outpost (A contract on an external chain) and a depot (contract on WNS core). It is a refinement of the protocol used by v0 of the LiqWNS system, but backwards compatibility is not implemented with that protocol.

The protocol defines asynchronous, bidirectional Assertion streams. The outpost transmits detailed information about its transactions and holdings, while the depot transmits instructions to the outpost for disbursing tokens and relays information about other chains. Separation of concerns between depot and outpost means that messages and local transactions can be interleaved in any order, so long as they are each processed sequentially.

In this context, transmission means storing/emitting messages somewhere readable to batch operators (e.g. contract events on Ethereum), who then convey the messages to the contract on the other chain

Design Overview

OPP's design centers on a hash-chained message format, where each message encapsulates a batch of assertions, a chaining header, and a payload checksum, ensuring integrity and order. The protocol supports multiple hash algorithms (defaulting to Keccak256) and flexible encoding. Message verification is achieved through a combination of batch operator submissions and independent verifier attestations, with mechanisms to handle out-of-order messages and echo lag.

Terms

Wire

The Wire SysIO blockchain

External Chain (or Native Chain)

A blockchain other than Wire, with which it is desired to transfer cryptographic assets/value.

Native Token

Any fungible or non-fungible token or other instrument which is tracked by

Outpost

A contract deployed on an external chain to facilitate movement of value into or through the wire chain.

Common State

The state data (contract holdings, user balances) of an outpost, which is replicated into the wire ecosystem.

Depot

A contract deployed on Wire that mirrors common-state data from one or more outposts.

Assertion

A single change to common-state, with a prescribed set of fields describing the change. One transaction may involve several changes.

Message

A collection of several assertions, with a timestamp and ID. Describes changes related to a single transaction.

Hash Chain

A sequence of messages together with hashed IDs derived from each message and the previous. Each hashed ID uniquely represents the entire history of the chain up to that point.

Protocol

Message Format

Chaining Header

The chaining header contains everything required to compute successive message IDs and verify message stream order.

Field	Type	Notes
from_chain	`chain_kind_t` (1 byte)	Enum `chain_kind_t : uint8_t` defined in `libfc-lite`
to_chain	`chain_kind_t` (1 byte)	Enum `chain_kind_t : uint8_t` defined in `libfc-lite`
message_id	32 bytes	The ensure lexigraphical correlation, the lsat 8 bytes are the sequence #
previous_message_id	32 bytes	
payload_size	4 bytes	
payload_checksum	32 bytes	
timestamp	8 bytes	Milliseconds since 1970-01-01 00:00
header_checksum	32 bytes	

Payload

The message payload consists of a header, followed by 0 or more assertions.

	Field	Type	Notes
payload header	version	1 byte	MSB == 0; range is 0x00 – 0x7F
	encoding_flags	1 byte	See Encoding Flags
	assertion_count	2 bytes	Number of assertions
repeat N times	assertion_type	2 bytes	
	assertion_length	2 bytes	encoded length of assertion
	assertion_data	bytes (variable)	

Each assertion type defines a payload format. These will have:

- Un-tagged fields – format is not self-describing
- Fixed field ordering
- Variable length fields (byte arrays, vectors) will be prefixed with varuint or uint32 length indicators.
- Optional or variant fields will be prefixed with a non-optional presence/type indicator, specified in the message definition.

If the message is little endian and varuint support is enabled, the format is identical to SysIO's `fc::raw`.

Hash Chaining

Procedure

The Message ID is produced using a chain-specific hash (see below) as follows:

1. If there is no previous ID or echo ID, use all-zero values.
2. Serialize payload header and assertions, hash to get Payload Checksum.
3. Serialize chaining header – all fields except Message ID.
4. Hash chaining header
5. Previous message number is extracted from first 8 bytes of previous ID as big-endian uint64
6. New message number computed by incrementing previous number
7. Serialize message number as big-endian uint64
8. Replace first 8 bytes of chaining header hash from step 4 with serialized message number

Hash Algorithm

No single hash algorithm is available to smart contracts on all chains at reasonable performance. Keccak256 has widest support so it is selected as the default. The protocol also allows sha256 and two others TBD.

Message Epochs

For purposes of cross-chain delivery, messages are grouped into epochs. When the end of an epoch is reached, a seal is added, with additional checksums and metadata, facilitating transport.

Epoch Clock

The originating contract uses a deterministic, on-chain clock (details vary by chain) to determine which messages belong to an epoch. When the clock shows an epoch has ended, before creating the first message of the new epoch, one or more summary messages are created at the end of the old epoch.

Epoch Crank

To facilitate the closing of the epoch, a crank function may be provided to finalize computation of checksums, merkle roots, etc. This function should be deterministic (no arguments), idempotent, and not require special permissions. Any reward paid to the crank turner is beyond the scope of this document. The contract should track `old_epoch` + `new_epoch`, to avoid interruptions due to crank timing, but if `new_epoch` expires before the crank is turned on `old_epoch`, it is acceptable to stop accepting transactions until crank turn.

Epoch Merkle

A Merkle root is computed of the sequence of message IDs contained in the epoch. This can be computed using an on-the-fly algorithm with total `n log(n)` time over `n` messages in the epoch, `log(n)` space, and `log(n)` final step time (see appendix). This allows a batch operator to do a partial delivery of epoch messages (if required due to transaction size or gas limits) by including a subset of messages together with a Merkle proof (see appendix).

Epoch Signatures

The epoch hash is computed from the serialized envelope, and this hash value is signed. The originating contract MAY collect signatures of the epoch, to include before transmissions (specifically the Epoch Hash is signed). The batch operator also adds their signature before transmitting.

Epoch Envelope

	Field	Type	Notes
	Epoch Hash	32 bytes	computed, not sent
hashed data	Epoch Number	uint32	
	Epoch Timestamp	uint64	== last message timestamp
	Previous Epoch Hash	32 bytes	
	Epoch Merkle	32 bytes	Merkle root of message IDs
	First Message ID	32 bytes	
	Last Message ID	32 bytes	
	Signatures	variable[]	chain-specific formatting

Message Processing

Message Storage

Each blockchain/family has unique features for persisting/emitting data, so this document cannot prescribe the exact mechanism used. Message must be stored in a manner that is immutable and that has sufficient de-facto persistence, meaning they will not be lost due to operator downtime, etc. It is not required that they be stored as part of the chain state – an un-trusted cache is sufficient: the batch operator can verify by constructing the chain and arriving at the current message ID stored on chain. Message storage need not be mutable or re-orderable. Epoch envelopes should be stored in the same way as the messages themselves.

Transmission

A batch operator follows the following procedure:

- Turn the epoch crank
- Retrieve epoch envelope
- Retrieve included messages, performing light validation checks.
- Add signature to envelope
- Determine whether the entire epoch can be processed at once
- If the epoch must be paged:
  - Divide messages into acceptably small pages
  - Compute merkle proofs connecting each page of messages to the envelope epoch merkle.
  - Submit envelope and signatures to destination contract.
  - Submit each page to destination, with merkle proof.
- If the epoch is not paged, submit envelope, signatures, and all pages; merkle proof not required.

Validation

Light Validation

The receiving contract, and any party adding a signature to the envelope should perform the following checks:

- Message numbers are sequential
- Epoch include first & last message
- Timestamps are monotonically increasing.
- Message numbers of Echo ID's are non-decreasing.
- Message parses successfully, and parsed length == indicated length.
- Re-compute Message IDs to verify chaining.
- Re-compute Epoch Merkle

Heavy/Oracular Validation

These checks require are expensive, difficult, or require knowledge not available from blockchain state

- Check that Echo ID's match the on-chain IDs from the counterpart contract.
- Check that processing of echoed ID's is correctly reflected in outgoing message data.
- Check transactions for the contract to ensure all are reflected in messages as appropriate.

Processing

Once a message is , it should be processed:

- Update mirrored data
- Update outgoing Echo ID
- Perform required actions (disbursing raw/liq or minting/assigning $WIRE)
- Depending on chain-specific implementation, storage of outgoing messages older than incoming Echo ID MAY be cleared.

Verification

A message epoch may be verified in one of two ways:

Multisig

In this method, a multi-sig (probably just a list of independent signatures, but a true multi-sig is also possible) is performed by, e.g., Wire block producers, collected by the originating contract before making the epoch envelope available for pickup by the batch operator. The destination contract must have a list of known keys for this signature, as well as a quorum rule for evaluating the multi-sig.

Optimistic + Challenge

When a message epoch is received from a known batch operator, it is held without processing for a challenge period (configurable). If no challenge occurs before the challenge period expires, the epoch is accepted and processed.

A challenge can start two ways:

- Forks: If two batch operators submit different epoch envelopes, a challenge is automatically started.
- Explicit Challenge: A challenger explicitly declares a submitted envelope to be invalid.

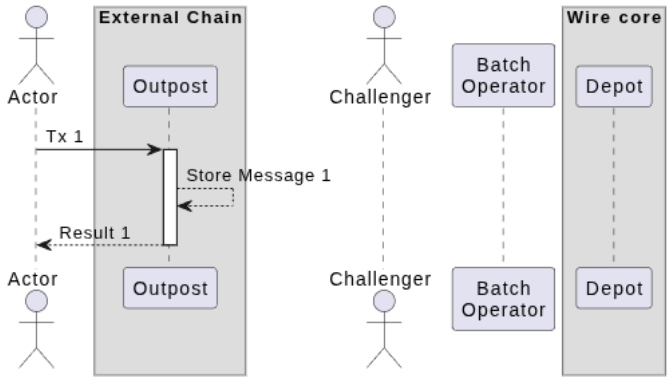
Once a challenge starts:

- A notification is sent to all challenger actors.
- Each challenger must call either `challenge` or `no_challenge`
- Contract waits for a majority (by collateral, threshold configurable) of challengers to report in, and for the original challenge period to elapse.

- Challenger votes are summed by collateralization, and the winning fork is accepted.

Flow Diagram

No longer accurate.

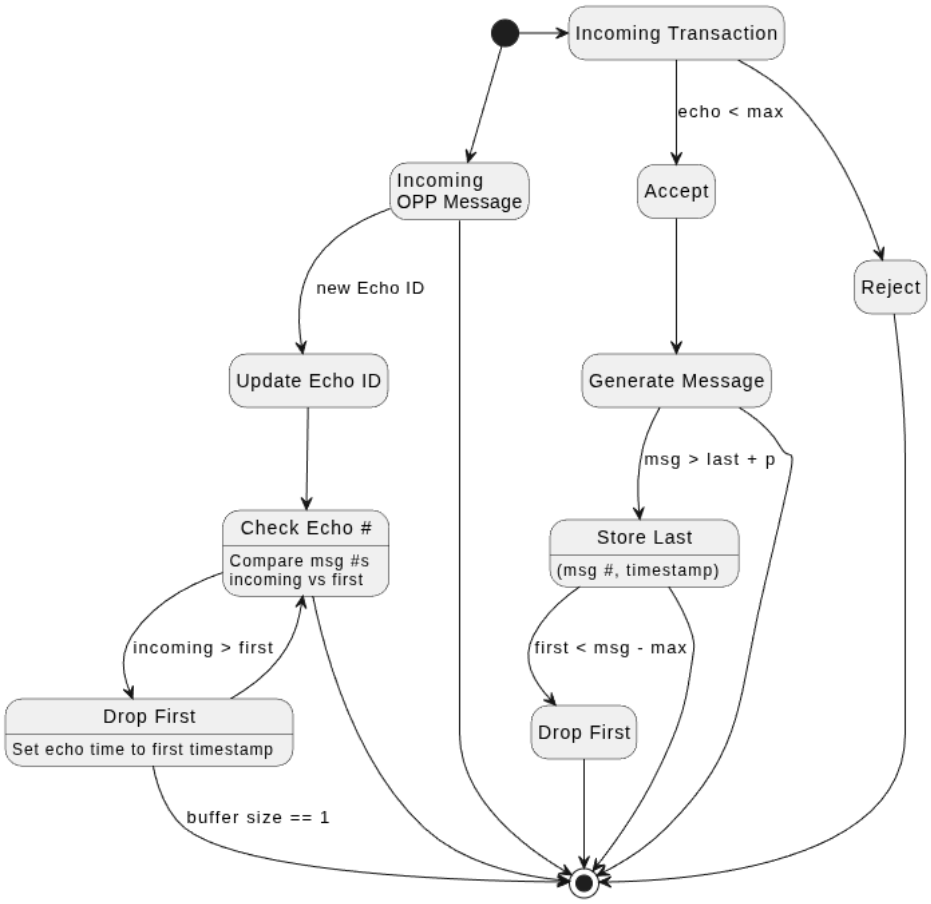


Echo Lag

The echo lag is the time or message count between the last message ID generated and the last echoed ID. An OPP contract may have a maximum lag, either message count or time, after which it will not accept local transactions that would generate messages.

Computing the count lag is trivial. If the contract does not have access to the full history of messages it has produced, A ring buffer or dequeue may be used, by storing pairs of message number and timestamp.

Diagram comments



- `msg` : Timestamp of current message (now)
- `echo` : Approx timestamp of last echoed message
- `first` : Timestamp of oldest ID in buffer
- `last` : Timestamp of newest ID in buffer
- `p` : precision – an acceptably large interval of time, smaller values mean more storage used.

Appendix

Encoding Flags

Mask (AND)	Result	Meaning
`0x01`	`0x00`	Big endian encoded
`0x01`	`0x01`	Little endian encoded
`0x06`	`0x00`	Hashed with keccak256
`0x06`	`0x02`	Hashed with sha256

`0x06`	`0x04`	*reserved – alternate hash*
`0x06`	`0x06`	*reserved – alternate hash*
`0x08`	`0x00`	varuint encoding used
`0x08`	`0x08`	uint32 used for length
`0xF0`	–	*reserved*

Assertions

Assertion definitions have moved here.

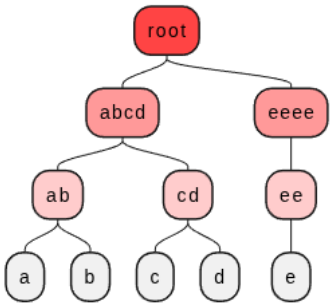
Protocol State

A contract implementing OPP should track the following state items:

Field	Type	Notes
Chain ID	2-4 Bytes	2-byte `ChainID` (Enum) - via a global enum supporting `65536` values 4-byte `ChainID` (Alpha numeric) - `[0-9A-Z]{4}` - supporting `1,679,616` values
Last Message ID	32 bytes	Last message emitted
Last Processed ID	32 bytes	Last message processed, becomes echo ID
~~Checkpoints~~	~~(32 bytes)[]~~	~~Message IDs which have been asserted as valid by verifier.~~
Incoming Messages	Tuple[Stamp, Message]	In-order stamp (Timestamp or any steady unit for measuring time) messages which fall into a time window covering at least `14` days.

		If stamp is `0` then the message has not yet been relayed.
~~Reorder Buffer~~	~~Message[]~~	~~[OPTIONAL] Pending out-of-order messages~~
Outgoing	Message[]	History of messages – chain-specific storage

Merkle Trees

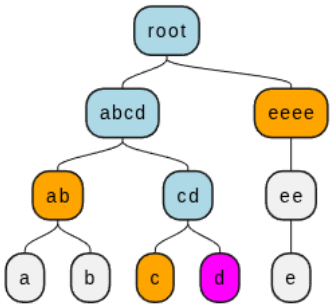


A Merkle tree is a collection of values (generally 32-byte hash values). It can be computed in an array by repeatedly hashing values together, pairwise, from left to right. If the final element is an odd number, it is concatenated to itself:

- [a] [b] [c] [d] [e] (5 elements)
- [ab] [cd] [ee] (3 elements
- [abcd] [eeee] (2 elements)
- [abcdeeee] root

Proofs

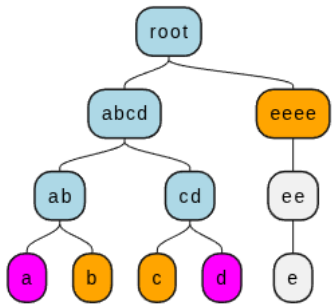
Inclusion of a single element can be proved by providing the sequence of other-side values at each combining step. Here is visualization of a proof of a single node (`d`):



In addition to `d` , three additional nodes must be included, `c` , `ab` , and `eeee` , together with an indicator of left or right combination:

- d: node to prove
- c: left
- ab: left
- eeee: right

It is also possible to generate a proof of multiple nodes at once:



Running Merkle

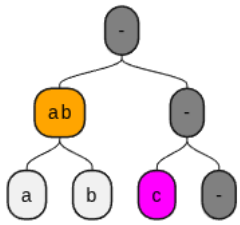
If we wish to compute a Merkle tree in a space-efficient way, we can compute interior nodes whenever the subtree descending from that node is complete. Consider the case where we have 2 leaf nodes, `a` and `b`. Placing these into an array, gives:

```
a, b
```

Since the ab subtree is complete, we can combine those leaves:

```
ab
```

When node c is received:

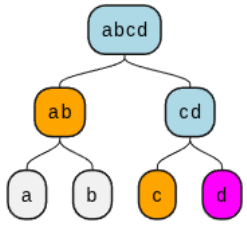


We now have the array:

```
ab, c
```

We cannot collapse further, because we do not know if the tree will terminate with `c`.

Suppose now we receive leaf `d`:



Our array will have:

```
ab, c, d
```

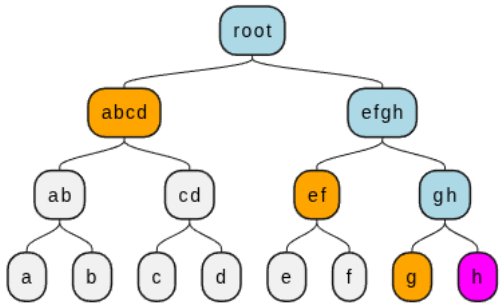
The `cd` subtree is now complete, so we can combine `c` and `d`:

```
ab, cd
```

But now we can go another step, because the abcd subtree is also complete, giving

```
abcd
```

Finally, let's look at what happens when leaf `f` arrives:



At this point, our array will contain:

```
abcd, ef, g, h
```

The `gh` subtree is complete, so we can combine those:

```
abcd, ef, gh
```

And the `efgh` subtree is complete:

```
abcd, efgh
```

But that is a complete subtree as well, so we collapse all the way down to a single value.

The reader will hopefully note at this point that, whenever the number of nodes *at a particular level* is even, it is possible to collapse the final two values in the tree. This leads us to the following algorithm (`merkle[-i]` is used to indicate elements relative to the end of the array, with -0 being the final element, -1 being the next to last, and so on) :

- Initialize empty array: `merkle[]`

- let `n` be the number of leaf added, initialized to 0
- when a leaf `x` is added:
  - set `merkle[n]` to `x`
  - increment `n`
  - set `m` = `n`
  - while `m` is even:
    - set `merkle[-1]` to `hash(merkle[-1], merkle[-0])`
    - drop the last element of `merkle[]`
    - divide `m` by 2

When the final leaf, `z` arrives, the tree is completed as follows:

- set `merkle[n]` to `z`
- increment `n`
- set `m` = `n`
- while `m` > `0`
  - if `m` is even:
    - set `merkle[-1]` to `hash(merkle[-1], merkle[-0])`
    - drop last element of `merkle[]`
  - if `m` is odd:
    - set `merkle[-0]` to `hash(merkle[-0], merkle[-0])`
    - increment `m`
  - divide `m` by 2