Transwarp-Conduit: Interoperable Blockchain Application Framework

Shidokht Hejazi-Sepehr, Ross Kitsis, and Ali Sharif

{*shidokht, ross, ali*}*@aion.network*

Aion Foundation

January 25, 2019

Abstract

Transwarp-Conduit (TWC) is a protocol for message transfers between two smart-contract enabled blockchain networks. Furthermore, we specify an application framework (leveraging the TWC protocol) that enables developers to define arbitrarily complex cross-blockchain applications, simply by

1. Deploying framework-compliant smart contracts and,
2. Hosting a TWC node (daemon process).

The TWC protocol is implementable without additional effort on part of the base blockchain protocol.[1]

1 Motivation

1.1 Scope

This work primarily focuses on public blockchains. Although vastly applicable, specific challenges pertaining to interoperability between private-to-private and private-to-public chains are out of the scope of this work.

1.2 Applicability of Work

We have observed a proliferation of public blockchains, with over 600 public blockchains in production as of end of 2018, with over 100 of these with a market capitalization of over $30M USD [2]. This diversity of well-capitalized public platforms reveals a limitation of blockchain technology: distinct blockchain constructions are required to address the vast problem-space.

It's not obvious if there will ever exist a sufficiently-generic blockchain platform, with enough transaction capacity, "to rule them all"[3]; a menagerie of specialized blockchains may become the status-quo.

In the interim, solutions are needed to interoperate between these purpose-specific and/or throughput-constrained blockchains, in order to broaden the scope and capabilities of applications that can be built on these platforms.

1.3 Use Cases

The ability for blockchains to access foreign blockchain-state (contingent on some finality criteria) makes possible a variety of applications which were primarily the realm of centralized solutions; notable use-cases include payment-versus-delivery, payment-versus-payment, cross-chain-oracles and other general-purpose inter-chain protocols for assets and information. Two of such use cases have been of particular public interest of late:

1. **Decentralized exchanges:** As of this writing, all major multi-crypto exchanges (conceptually) consist of a centralized server responsible for taking and clearing orders. Interoperability infrastructure enables cross-chain exchanges to be built in a completely decentralized way: smart contracts on supported chains coordinating to take and clear orders without centralized intervention.

[1] This assumes smart contract language expressivity and availability of cryptographic primitives common to most blockchains which are categorized as smart-contract enabled.

[2] Data sourced from coinmarketcap.com on December 30, 2018; used the "Not Mintable" and "Market Cap" properties of reported cryptocurrencies to construct this claim.

[3] There seems to be an arms race today between several well-capitalized public blockchain organizations (Ethereum, IOHK, Parity, EOS, etc.) to invent a public blockchain platform "to-rule-them-all", with diverse hypotheses on scaling.

2. **Portable assets:** The ability to issue assets that can be transported between blockchains without violating the properties of said asset (fungibility, ownership, value, etc.) has been of great interest to create productive (smart-contract-based) economical systems and legal frameworks spanning multiple blockchains.

2 Interoperability Definition

So far, we've invoked the term "interoperability" without much precision; let's precisely define the term now.

When speaking about interoperability, we are concerned with establishing a causal relationship between events on two blockchain (isolated systems). Two[4] possible models exist to establish a relationships between events which need to be correlated (the task of interoperability) [1]: causation and dependency.

Assume we have two blockchains B_A and B_B, that must interoperate:

1. **Causation**: some internal event on one blockchain *causes* a correlated internal event to occur on another blockchain.
   (a) ***Forward causation***: event X on $B_A \Rightarrow$ event Y on B_B.
   (b) ***Reverse causation***: event V on $B_B \Rightarrow$ event W on B_A.
2. **Dependency**: some external event *causes* a pair of internal events to occur on both B_A and B_B (i.e. both blockchain events correlated by the same external event).
   (a) E.g. external event Z $\Rightarrow$ (event X on B_B AND event Y on B_A).

From the perspective of certain applications, a dependency relationship may be sufficiently expressive (e.g. an oracle publishing the same data stream to multiple blockchains). But most interoperability relationships, particularly the ones described in section 1, require the structure of causation for their articulation. For the purposes of this paper, when we say:

"Two blockchains interoperate",

we mean:

"There exists a mechanism to establish causation between transactions sealed onto distinct blockchains[5]".

3 Design-Space

This section outlines the trade-offs available to the authors with respect to available interoperability architectures.

A multi-signature notary scheme was the implementation-strategy of choice for the TWC protocol, since it provided a reasonable trade-off between decentralization (by opting into a well-defined trust model), ease-of-implementation, system reliability and on-chain data-amplification.

Three families of interoperability mechanisms were identified for analysis:

1. Time-locks (hash locks)
2. Relaying (validate-one-chain-inside-another)
3. Notary schemes

An exposition of each of these methods is provided to highlight key design trade-offs; a more nuanced discussion of these mechanisms is available in [1] and [5].

3.1 Time Locks (Hash Locking)

These systems comprise of some time-locked operation across two chains, which triggers on the revelation of the pre-image of a one-way function. We illustrate this concept with the well-known example of hash locking (a.k.a. "atomic swaps"):

Assume entities X and Y need to swap some asset that exists on two distinct chains (B_1 and B_2). Assume some asset C_1 is owned by X on B_1 and asset C_2 is owned by Y on B_2. Furthermore, users X and Y have accounts on both blockchains (X_{B1}, X_{B2}, Y_{B1}, Y_{B2}). There exists some well-defined time interval T. Also assume some ideal hash function H.

1. X generates a secret S and generates a hash H(S) and sends H(S) to Y.

[4] Coincidence is a third possible relationship, but it can be safely ignored since its likelihood is heuristically understood to be minuscule.

[5] The blockchains have different genesis blocks and share no history.

2. X_{B1} locks asset C_1 on B_1 with the following conditions:
   (a) If S is revealed within time 2T, C_1 is transferred to Y_{B1},
   (b) Else, C_1 is transferred back to X_{B1}.
3. After Y sees X lock asset C_1 on B_1, Y uses their account on B_2 (Y_{B2}) to lock asset C_2 with the following conditions:
   (a) If S is revealed within time T, C_1 is transferred to X_{B2},
   (b) Else, C_2 is transferred back to Y_{B2}.
4. X reveals secret S on B_2 to claim C_2.
5. Y observes S on B_2 and publishes it on B_1 to claim C_1.

3.1.1 Analysis

Several implementations and discussions of this scheme exist [2, 3]; the discussion on this topic here is limited in so far as it illustrates the following relevant facts about this system:

1. The protocol is only capable of defining a *dependency* relationship between events on the two blockchains (B_1 and B_2); in our illustration, the asset transfer events on B_1 and B_2 are dependent on the revelation of the pre-image of the hash.
2. This protocol requires a high degree of interactivity; each user (X and Y) is required to interact with both blockchains at least once, in addition to off-chain (real-time) requirements of observing and reacting to events on a blockchain.

3.2 Relaying

This mechanism relies on the destination chain B_D, validating the consensus rules of the source chain B_S (making the destination chain effectively a "light-client" of the source chain). An event (E_D) on the B_D can now depend on an event (E_S) on B_S through light-client verification techniques available for the B_S (e.g. providing a Merkle-proof for E_S's membership in a block on B_S). Let's illustrate this with an example:

Assume two blockchains, source chain (B_S) and destination chain (B_D), which both have the following properties:

1. They have the notion of the block headers, which is a data structure whose hash can uniquely identify a block [7].
2. The block headers are some finite, well-defined size that is much smaller than the full block contents.
3. They collect all transfers included in a block into a Merkle tree, and then store the root of that tree in the block header.
4. The consensus rules can be validated completely by examining the block headers.

Now let's construct a smart contract R_D that has functions that:

1. Accepts block headers from the source blockchain B_S, validates the consensus rules of the source blockchain and maintains a "light-replica" of B_S in the form of a chain of block headers.
2. Accepts a transaction hash, a Merkle proof and a source block number, and validates that the transaction was sealed in the source chain.

Now one could imagine constructing a smart contract on B_D that leverages our relay contract R_D to accept proof-of-inclusion of some transaction that occurred on B_S before executing its logic. This effectively allows one to establish a *forward causation* relationship (section 2) between B_S and B_D.

3.2.1 Analysis

One powerful property of this system is that it's completely trustless:

1. The relay contract R_D enforces the source chain's consensus rules in a trustless smart contract (publicly auditable)
2. Anyone can run a process to relay (hence the name) the source chain (B_S) block headers to R_D. Notice that liveness of the whole system is dependent on at-least one relaying entity being online.

But there are some problems with this system:

1. The amount of data required on the destination chain is proportional to the size of all the source chain's block headers since its origin block. This can be problematic due to block size limitations on the destination chain. This can be further exacerbated by an impedance mismatch between the chains; if the source chain's block time is much faster than the destination blockchain, then the size of the source block headers per destination block can become a significant proportion of the block limit. Several solutions to this problem have been proposed in [6] and [4], and are actively under investigation.
2. This scheme requires the re-implementation of the consensus algorithm in the smart-contract language of the destination chain. In a world with diverse blockchains, there are several problems with this approach:
   (a) Blockchain platforms use a diverse range of cryptographic primitives to validate their consensus algorithms; most smart contract languages currently have limited support for cryptographic primitives other than the ones used natively.
   (b) Even if all the required primitives to validate the diversity of consensus algorithm implementations were universally available, re-implementing the same algorithm across a non-trivial number of platforms carries significant operational risk and costs.

In the opinion of the authors, exclusive usage of relaying to power a flexible interoperability solution requires further research and system design.

3.3 Notary Schemes

In this scheme, some external (notary) entity is allowed to make claims on one blockchain about what happened on another blockchain. For this to work, the blockchain must trust the notary entity to honestly make claims about some other external blockchain.

Since a primary design goal of blockchains is decentralization, a combination of several strategies can be used to mitigate fragilities which arise from trusting a single external entity:

1. Use a set of well-known entities, who are disincentivized to deviate from protocol, either via reputation-at-stake mechanism or bonded financial resources.
2. Force the notary entities to put up financial stake, which they can stand to lose if they misbehave.
3. Have a plurality (some appropriately large number) of signatories, who need to agree on the same value, either interactively via a consensus algorithm, or non-interactively via a multi-signature scheme.

3.3.1 Analysis

This scheme offers the most versatility in establishing causation relationships between transactions on two blockchains with diverse structures and capabilities; since the relay and signatory processes can be implemented in any general-purpose language, they serve to translate between two blockchains. Let's illustrate this idea with an example:

Assume two distinct, interoperating blockchains $B_1 \rightarrow B_2$ use different asymmetric cryptographic systems (Ω_1, Ω_2) and one-way functions (H_1, H_2). Since the signatory processes are implemented on some general-purpose computation platform (e.g. linux on AMD64), there is no restriction on the crypto-system which can be implemented therein. The smart contract implementation on B_2 only needs to understand Ω_2; the signatory can cryptographically attest to its observations of B_1 using Ω_2.

3.4 Architecture Selection

At the time of authorship of the TWC framework report, a notary scheme with a well-defined trust model was considered the best path to a production interoperability solution in the very short term; absolute decentralization was traded-off against its flexibility in enabling a uniform interoperability solution across a diversity of blockchains.

Hash locking was precluded from consideration on part of its inability to express causality relationships between chains.

Although the relaying scheme offers very attractive decentralization properties, write amplification and limitations of today's smart contract platforms make a uniform implementation across a diversity of blockchains infeasible in the short term. As capabilities of smart contract platforms improve, this will become a more compelling architecture for interoperability.

4 System Overview

4.1 Design Objectives

The following objectives were adhered-to when making system-level design decisions:

1. **Keep-it-simple**: a simpler protocol is heuristically less error-prone and easier to reason about.
2. **Favor safety**: the protocol should favor safety (primarily when trade-off is against liveness).
3. **Minimize interactivity**: the protocol should require minimal interactivity from the end user; it should make no assumptions about the online-ness of the user of the protocol, beyond those made by the blockchain where a transaction originates.

4.2 Protocol Description

This section provides an overview of the (a detailed description is available in section 5).

For the purposes of this discussion, this protocol is modelled as a message passing mechanism between a source blockchain and a destination blockchain. Note that the protocol is bi-directional; the unidirectional case is presented for expository purposes. The reverse information flow is simply a dual (symmetric in the other direction).

The simple construction is as follows:

1. A message M gets sealed into block N_S on a source blockchain B_S.
2. Wait for F_S blocks to elapse on source blockchain B_S (F_S being the acceptable finality on source chain).
3. At some time ε after block number $N_S + F_S$ is sealed on B_S, message M gets sealed into block N_D onto the destination blockchain B_D. (Note that ε models the time required for the "message pasting action" to take place).
4. Wait for F_D blocks to elapse on destination chain B_D (F_D being the acceptable finality on the destination chain).

The above construction contained nothing about how the system works, merely what it does. In order to describe how the protocol works, we first outline the actors involved and their responsibilities; we then summarize the interactions between them.

1. Blockchain (B_S, B_D): source and destination blockchains respectively, for some message M.
2. Signatories (S_n, n > 0): passive entities, which upon the request of the relay, make independent observations on source blockchain (B_S) and attest to the
   (a) History (when M was sealed onto B_S) and,
   (b) Contents of M.
3. Relay (R): coordinates the flow of the message from source to destination chain:
   (a) Actively monitors B_S for any messages addressed to be replicated on B_D.
   (b) Collect signatures on the history and content of M from 2/3 of S_n.
   (c) Send a transaction to B_D containing M.

We need one more bit of exposition, before we're ready reveal the protocol in its entirety. So far we have been suspiciously vague about the nature of the message M, so let's define that now.

The system assumption we made at the outset of this document is that the blockchains are being "bridged" must:

1. Have a sufficiently expressive smart contract language to define arbitrary user-defined functions, and
2. Must implement language primitives necessary to validate signature (the signature algorithm can be anything).

The message M at the source blockchain is a transaction containing message M, that is the successful execution of the "egress" function *requestTransfer(message)* on a smart contract we refer to as the "Adapter" well known to Relay (R). The implementation of the *requestTransfer(message)* function must adhere to the specs defined in section 5.

The message M at the destination blockchain is a transaction that consumes the message generated from the egress function of the Adapter contract deployed on the source blockchain. In particular, the destination smart contract must implement the "ingress" function *processTransfer(message, signatures)* as defined in section 5.

In particular, the message M is composed of two features:

1. History: when M was sealed onto B_S
2. Contents: a binary payload encoding the desired function call and arguments on the destination chain (as defined in section 5)

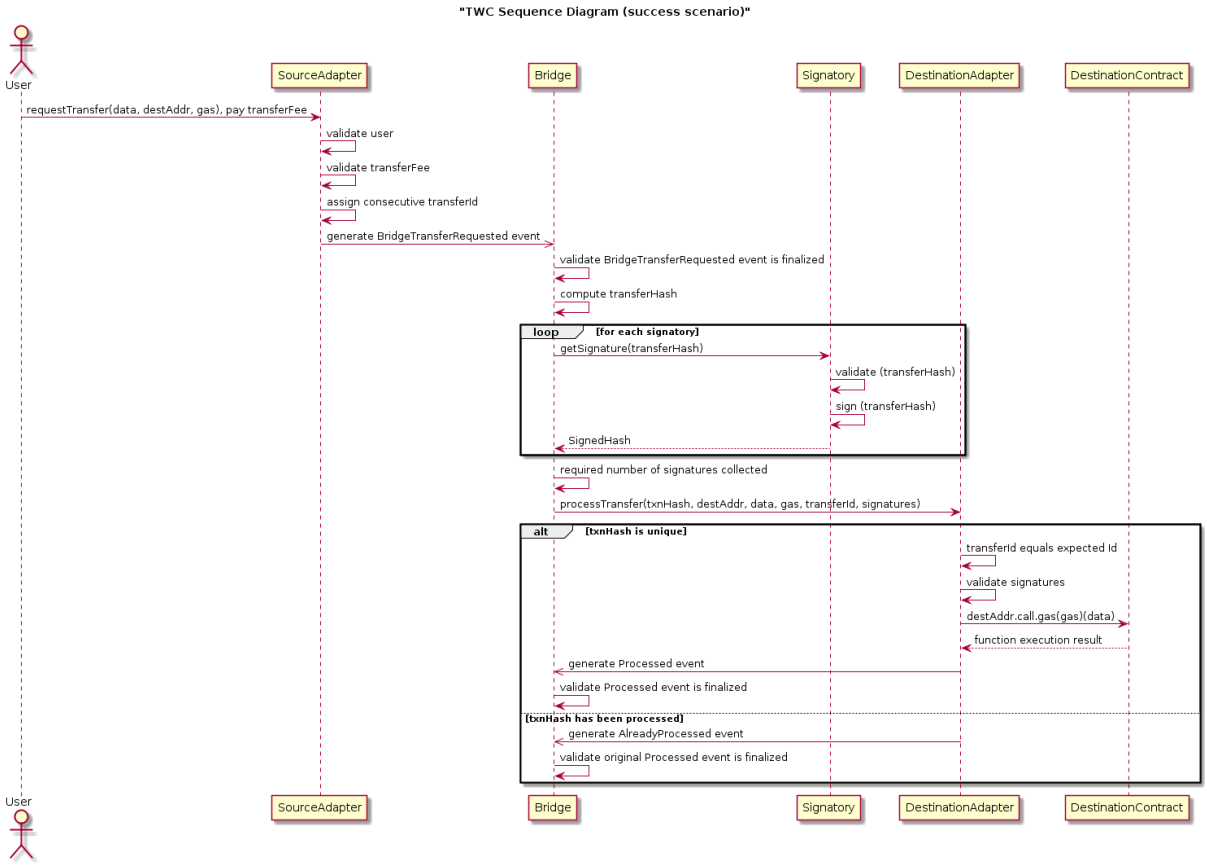

Figure 1: Summarizes the "happy path" of the protocol in its entirety.

4.3 Trust Model

The trust model is implemented entirely in the destination smart contract. The destination smart contract must implement the following rules:

1. The smart contract must have access to a pre-defined list of N_{sig} Signatories (public keys) whose signatures on the history and contents of M it implicitly trusts.

2. The smart contract must define a Relay (address) which is the only entity allowed to post ingress messages to the contract.
3. The ingress function *processTransfer(message, signatures)* shall implement the following rules:
   (a) At least 2/3 of N_{sig} signatures are required before message is processed.
   (b) Only the Relay is allowed invoke this function.

As it's obvious from the construction, the entire trust model is encoded in the destination smart contract. A quorum of signatories is responsible for attestation of the source message; i.e. as long as a two-thirds majority of the signatories agree that message M was observed on the source chain, the destination chain accepts that as fact and proceeds to process M.

Furthermore, in this model, the Relay entity (R) is trustless; the dependency in the trust model exists only as a first-order implementation of a "paranoid fencing" system a service-provider might implement to ensure contract invocation by trusted accounts to prevent public exploitation of bugs discovered in the production smart contract. This framework is not opinionated in the incentive mechanisms for the Relay to continue providing service; these incentives can be implemented in an application-by-application basis (at the smart contract level). The cost of the transfer is borne by the Relay entity, which serves as a built-in denial-of-service deterrent.

4.4 Safety and Liveness

We provide a crude sketch for the safety and liveness arguments for this system.

4.4.1 Safety Sketch

For the purposes of this sketch, we use the heuristic definition of safety: "nothing bad will ever happen". The safety of this system is dependent on the set of signatories and the smart contract implementation. The safety of the system can be guaranteed if:

1. At-least two-thirds (2/3) of the signatories make faithful observations on the source blockchain B_S's main-chain, and
2. If all protocol constraints defined in section 5 are implemented correctly in the ingress function on the Adapter contract on the destination chain (B_D).

4.4.2 Liveness Sketch

For the purposes of this sketch, we use the heuristic definition of liveness: "something good will eventually happen". The liveness of the system is dependent on the set of signatories, the relay and the blockchains involved. The liveness of system can be guaranteed if:

1. The liveness of the blockchains being "bridged" is not violated,
2. At-least two-thirds (2/3) of the signatories respond to requests for signatures from the Relay within some pre-defined time (synchrony),
3. Some entity is providing a Relay service with liveness guarantees.

Notice that the relay is only responsible for liveness; it needs to be always online and maintain at least one connection to each of the blockchain clients being "bridged". If the relay is byzantine and the signatories are not, the system comes to the halt (i.e. messages are not replicated on the destination chain) and no malicious messages get delivered to the destination chain.

A more in-depth discussion of the failure and liveness of the system is available in section 6.

5 Protocol Specification

This section provides a high-level description of the TWC operation and usage. The TWC is split into 4 components:

1. Developer defined smart contract
2. Adapter contract
3. TWC bridge node
4. TWC signatories

5.1 Developer Defined Smart Contract

This smart contract is any contract or application defined by the developer; the TWC places no restrictions on the functionality or implementation of the contract. Initial use cases are expected to be cross chain asset movement, however, the TWC is a generic protocol allowing for the transfer of any arbitrary data; not just that of representing assets.

Any interoperable smart contract must implement a function to form the ABI encoded data and call *requestTransfer* of its corresponding Adapter. Optionally, the source smart contract should contain developer defined external remediation functionality. The security model section 6.1 provides details for scenarios which may cause inconsistencies between the source and destination chains.

5.2 Adapter Contract

The Adapter contract is the primary interface between the TWC node and the developer defined smart contract. At the very least, the Adapter should be initialized by providing a valid set of signatories, Relayer account address, minimum number of required signatures and cross-chain transfer fee. Additionally, an authorized set of senders should be provided if the request initiators are to be validated. The core functionality of the Adapter is to:

1. Accept incoming messages (encoded function calls) from developer smart contracts or applications through *requestTransfer*.
    (a) Assign a nonce to each transfer and charge usage fees as specified by the owner. Optionally, it may also verify senders, preventing unauthorized users from initiating cross chain transfers.
    (b) Emit outgoing message events to be picked up by TWC (*BridgeTransferRequested*)
2. Accept incoming messages from the TWC node (identified by the Relayer account) through *processTransfer*.
    (a) Verify transfer signatures against the current set of signatories and threshold.
    (b) Maintain a list of processed messages and reject duplicate messages.
    (c) Track nonces of previously sent messages and reject out-of-order transfers.
    (d) Call developer smart contracts.
    (e) Emit events indicating the transfer status (*Processed/AlreadyProcessed)*.

Detailed functionality and source code of Adapter contracts is provided in Appendix C.

5.3 TWC Bridge Node

The TWC bridge node is the primary component connecting two distinct blockchains and it is responsible for the following:

1. Monitor connecting chain status
2. Monitor Adapter smart contract events
3. Process transfer requests
    - Extract transfer data from source chain events
    - Maintain transfer order based on a sequential nonce
    - Request signatures
    - Send transfers to the destination chain
    - Monitor transfer status

5.3.1 Monitoring Connecting Chain Status

The TWC node must monitor the status of both B_S and B_D. Monitoring the status of a blockchain will vary depending on chain properties, thus specifics are left to the implementation. In a general sense, chain status must encompass at least *liveliness* and *reorganization*. Liveliness refers to the generation of new blocks, while reorganization events lead to a chain's history being re-written or switching to a different branch.

In addition to monitoring chain status the TWC must also respond when detecting incorrect behavior. Responses will vary by implementation however there are 3 strategies that may be followed:

1. Pause operation until operator intervention
2. Retry operations as needed
3. Continue as normal

5.3.2 Monitoring Adapter Smart Contract Events

In order to detect transfer requests and responses, TWC node must monitor each connecting blockchain for events generated by the Adapter contracts. From the source Adapter the TWC must monitor the following events:

- **BridgeTransferRequested:** Specifies a new transfer

From the destination Adapter the TWC must monitor the following events:

- **Processed:** Indicates a transfer has been processed.
- **AlreadyProcessed:** Indicates a submitted transfer was previously processed and has been resubmitted.

5.3.3 Processing Transfer Requests

Processing transfer requests is the primary role of the TWC node and it is performed in the following steps:

1. Detect a *BridgeTransferRequested* event from B_S.
2. Extract transfer details from the transfer request.
3. Create a signing request.
4. Send signing request to all S_n for verification and signatures.
5. Collect signatures and create a transfer transaction.
6. Submit the transfer to B_D.
7. Monitor B_D to ensure the transfer transaction is successful.
8. Listen for *Processed* or *AlreadyProcessed* events from the Adapter smart contract.

In addition to processing transfer requests, TWC nodes must also provide a guarantee on transaction ordering i.e. the TWC must process transactions in the same order as they occur on the source chain.

5.4 TWC Signatories

The primary function of the signatories is to process signature requests, validate transactions and finally sign valid transfers. Transfer sign requests contain the minimal amount of data needed for verification:

- Source chain block number
- Source chain block hash
- Source chain transaction hash
- Source chain transaction data hash

Using the block number, block hash and transaction hash signatories must:

1. Fetch the specified block in order to ensure their view of the chain is the same as that of the TWC node.
2. Fetch the specified transaction, take a hash of transaction data and compare it to the provided transaction data hash.
3. Given the provided transaction data hash and calculated data hash are equal, the signatories will sign the transaction data hash using their private key.

TWC nodes collect these signatures in order to attest to an event occurring on the source chain.

6 Failure Modes

Proof-of-Work (PoW) blockchains as a standalone data structure and application share several common attack vectors and behaviors, these include:

1. 51% attacks leading to reversion of transactions.
2. 51% attacks leading to transaction censorship.
3. 51% attacks leading to the creation of an invalid chain which is detected by full nodes. However, light nodes may accept the invalid chains.
4. Soft forks.
5. Hard forks (bulk of the user bases switches to the new chain).
6. Network splitting into multiple fragments.
7. Consistency favoring chains halting due to a network issue.

Any one of these events may cause serious issues for a single chain; interoperability between chains amplifies the effects of each of these events. Cross-chain application developers must be cognizant of chain failure events and build chain failure checks into their applications. Even when handling chain events within an application developers and operators must be ready with off-chain solutions; certain cases detailed in the following section may not be detectable until it is too late to stop the invalid cross chain transaction. These cases will require an off-chain solution to correct the invalid state introduced on the destination chain.

During exceptional circumstances chains may become compromised prompting emergency responses such as the DAO or bitcoin overflow fork. The TWC significantly complicates these types of forks, once a message has been sent across the TWC, a fork on the source chain cannot reverse transactions on the destination chain. These types of events have the potential to lead to attacks such as double spending of transferred assets on the destination chain. While these types of events are extraordinarily rare, developers and operators should be aware of the impact of these types of events.

Given the current state of blockchain technology; without radically changing blockchain implementations a fully decentralized, failure proof solution is impossible to achieve. Some sort of trade-off between risk, centralization and interactivity must be made by interoperability solutions, broadly these may be defined as:

- **Centralization:** The level of control a single actor may exert over a system; given a fully centralized system a single actor may fully control cross chain functionality while in a fully decentralized system no one actor may control cross chain functions.
- **Risk:** The level of risk operators are willing to accept. A higher level of risk may simplify operation and increase throughput, however, chain events may be exploited to create invalid states on the destination chain.
- **Interactivity:** The level of interaction required by users in order to validate bridge operation. As the level of interaction increases, users are always expected to be online and monitor the operation; should users be offline invalid transactions may be submitted and accepted on the destination chain.

The TWC and notary-based schemes in general choose to trade centralization for interactivity. Very little interactivity beyond initiating the transfer is required from the user. However, to achieve the low level of interactivity the TWC must inherently trust its set of signatories to validate and attest to the validity of transfers.

The remainder of the section focuses more specifically on the security model and vulnerabilities in the described notary based TWC bridging solution.

6.1 Security Model

The security model of the TWC is provided by the signatories making faithful observations of the source chain as well as the Adapter contracts. Given these constraints, no single compromised system component may compromise the entire system.

6.1.1 Risk and Impact

Before exploring the TWC security model a ranking is assigned to the risk of exploiting an attack vector and the potential impact a compromised party may have on the TWC. Impact is examined from the view of contracts entering invalid states. Propagation of invalid transactions to the destination chain exponentially increases the damaging impact of an attack.

Risk levels:

- Low: Attack requires multiple independent parties to be simultaneously compromised.
- Medium: Attack requires a single party to be fully compromised.
- High: Attack may proceed without any parties fully compromised.

Impact levels:

- Low: It may temporarily shut down a bridge and cause inconvenience. No data will be corrupted on either B_S or B_D and no external remediation is required.
- Medium: One or more bridges are totally censored and unable to transfer transactions. No corruption of contract states; minor remedial actions may be required mainly in configuration changes.
- High: B_S or B_D contract state corrupted. Possible double spend attacks, theft or complete contract state invalidation. Remediation will likely be required by the contract owner (if possible).

6.2 Single Party Compromise

6.2.1 Source Chain Infrastructure (Risk: Low, Impact: Low-High)

A compromised source chain is defined as at least 2/3 of connections to B_S providing faulty data, causing the bridge to detect and process incorrect data. The impact of such an attack may be broken up into two scenarios:

- Individual nodes return incorrect data; however, the chain itself is intact and signatories follow the original chain. In this scenario, there will be no impact on the destination chain; signatories will reject the invalid transactions and they will not be submitted to the destination chain.
- Signatory chain connections may also return invalid data. This may cause the destination contract to reach an invalid state. Such a scenario is equivalent to multiple parties or the entire source chain being compromised, and will be explored further in section 6.3.

6.2.2 Destination Chain Infrastructure (Risk: Low, Impact: Low)

A compromised destination chain is defined as at least 2/3 of connections to B_D providing faulty data. The impact of this attack is quite low as actions are triggered by the source chain. At worst an attacker may be able to stall the bridge or cause it to shut down due to inconsistent messages. Attacks of this nature may be mitigated by moving to alternative chain connections where bridge operation may resume.

6.2.3 Adapter Contract (Risk: Medium, Impact: High)

The Adapter contract is a critical piece of the TWC, thus compromising it on either the source or destination chain may have quite serious consequences. In order to compromise an Adapter, an attacker needs to gain access to its owner's private key. Adapter owner account can modify important properties such as authorized senders, signatory public keys and Relayer address. Combining a subset of these changes will allow an attacker to corrupt the state of user contracts on either B_S or B_D. Attacks could be performed on source or destination functionality.

One possible attack on the source side would be to modify the authorized sender list to allow an attacker to use their own contract to generate transfer requests in the place of a user's contract. This will allow attackers to bypass a user's defined logic by substituting their own. The TWC node and signatories have no way of distinguishing the original source of the transaction, thus the transfer will appear valid. The message will be sent to the destination chain and accepted by the destination Adapter; setting an invalid state on the destination contract.

A possible attack on the destination would modify the set of signatory public keys as well as the Relayer account; giving an attacker the ability to send any message to the destination contract, completely cutting out the TWC node and signatories. Invalid messages would have the ability to set an invalid state in the destination contract.

Attacks of this nature are less TWC specific and are more a generic class of attacks on all smart contracts; once an owner's private key has been compromised it is inevitable that the attacker will gain a large amount of control over the contract; which he may exploit in malicious ways. The extent of the damage an attacker could do in the context of TWC is difficult to quantify as different contracts will have different levels of access.

The TWC protocol spec regarding Adapters mandates each change in the contract to generate a corresponding event. While the spec does not specify monitoring and usage of those events, a TWC node or an external process could be set to monitor them. If an unexpected change is detected, the monitoring process could take emergency measures such as shutting down the TWC to ensure those changes do not allow cross-chain attacks.

6.2.4 Bridge Node (Risk: Medium, Impact: Low)

A compromised bridge may perform 4 types of malicious behaviors:

1. **Request signatories to sign invalid transactions:** Signatories independently verify each signing request; without locating and verifying the corresponding transaction on the source chain, a signatory will not sign the request. Therefore, sending invalid transactions for signatures will not impact bridge, source or destination chains.

   Stemming from this type of attack is attempting to flood signatories with invalid signature requests. Attacks of this nature may be mitigated by rate limiting each connection. A malicious bridge may max out its own connection, however, this will not negatively affect other bridges.

2. **Send invalid transactions to the destination chain:** The destination chain Adapter will verify each signature based on its configuration. As the bridge is unable to falsify signatures, the destination chain will detect invalid transactions and reject them.

3. **Censor transactions:** All events are assigned a nonce (*transferId*) by the source Adapter contract allowing the destination Adapter to detect missing transactions and reject out of order nonces. Attempting to censor a transaction will lead to a bridge stalling as the destination chain will not process transactions until the expected nonce is submitted. While this will stop the following legitimate transactions, we believe it is an acceptable trade-off since it will immediately alert the bridge operator and users of incorrect behavior, allowing them to act accordingly.

4. **Replay transactions:** A compromised bridge may store signed transactions and replay them later, without having to request signatures from signatories. Destination Adapter maintains a history of all processed transactions; hence, replayed transactions will be rejected by the Adapter.

6.2.5 Signatories (Risk: Low, Impact: Medium)

A compromised signatory set is defined as at least 2/3 of S_n censoring transactions; either by refusing to sign validation requests or by providing incorrect signatures.

- **Refusal to sign:** In this scenario signatories will simply stop replying to bridge requests, allowing them to time out. A bridge will not be able to distinguish this case from a network error and will respond in the same fashion. Without noting specific implementation details the only option a bridge will have is to re-try sending the message, either indefinitely or for some period before giving up on the request. The state on the destination chain will not be corrupted.

- **Incorrect signatures:** In this scenario signatories will reply to validation requests, however, the replies will contain invalid signatures. Bridge performs cursory checks on signatures, but since it is unaware of the specifics of a signature, it is unable to verify it. If the signatures pass all cursory checks, the bridge will forward the transfer to the destination chain. This transfer will fail the signature validation step in the Adapter contract and the state will not be modified.

In both scenarios the state of the destination contract will be protected by either the bridge or the Adapter contract. Messages on the source chain may be stuck in a pending state while signatory functionality is restored; however, no messages will ever be lost. While inconvenient, bridge operators may switch the signatory set to begin processing backlogged messages or simply wait till the signatory behavior is corrected.

6.2.6 Operator (Risk: Low/Medium, Impact: Medium/High)

An operator is a special actor within the TWC owning a significant share of the TWC infrastructure, as such the operator has a great deal of control over the TWC. Due to the high level of access in the user defined contract, the operator is generally seen as controlling the TWC node and Adapter contracts; in most cases they will also control the user contract however this is not strictly required. Consequently, an operator will have access to the following:

- Relayer account keys
- Adapter owner keys
- User contract owner keys
- TWC node operation

Ideally the operator's role would be split among at least 3 entities: relay owner, Adapter owner and user contract owner. Operators are strongly encouraged to use separate keys for deploying their smart contracts and setting up the Relayer account. The exact level of risk and impact for this scenario largely depends on the deployment standards followed by the operator; a simple change in account management stops a single compromised account from cascading and compromising the entire system.

An important aspect of this threat model is that operators are simultaneously incentivized to secure their keys and disincentized from using the TWC to performing attacks on their own contracts. This is analogous to mining pools performing attacks on their own network; doing so would simply destroy the pool's reputation and income stream. Mining pools are encouraged to act honestly and secure their operation to ensure an attacker could not gain access. Similarly, operators are encouraged to act honestly as exploiting their access to attack their own DApps would destroy their reputation and application.

An attacker will have limited capabilities in the scenario of an operator's keys becoming compromised; given a separate set of keys were used for each deployment.

- Relayer account: The set of actions an attacker could perform is a subset of those an attacker could perform in compromising the TWC node. Ultimately the Adapters will identify and reject any invalid messages.
- Adapter owner account: Compromising the Adapter account is considered in section 6.2.3.
- User contract: This attack vector is outside of the scope of the TWC, attacks will largely depend on functionality in the user contract.
- TWC Node: Compromising a TWC bridge node is considered in section 6.2.4.

The scenario of an operator's keys becoming compromised where the key is used in multiple deployments is equivalent to a multi-party compromise (considered in section 6.3).

6.3 Multi-Party Compromise

The inherent trust model of the bridge and signatories in a notary model may result in multi-party compromise leading to catastrophic errors on the destination chain. Due to the high number of possible pairings only one of the most serious scenarios will be covered.

6.3.1 Bridge and Signatories (Risk: Low, Impact: High)

The destination chain relies on the bridge and signatories as its source of truth about the source chain. Given a compromised bridge and signatory set, a bridge may create any transaction; regardless of occurrence on the source chain. Compounding the issue, the compromised signatory set will sign the invalid transactions. Once the invalid transaction is transmitted to the destination chain, from the point of view of the destination contract, the transaction will be deemed valid. The destination chain will successfully validate the signatures and modify the contract state to an invalid state as specified in the falsified transaction.

Consider a transfer of value from a compromised bridge and signatory set; attackers may mint any number of tokens on the destination chain. Once minted, these tokens are indistinguishable from real tokens. Assuming enough liquidity exists, an attacker may transfer their tokens back to the original chain and sell their newly minted tokens. An attack of this nature may modify the total supply of a token; possibly fundamentally breaking the original smart contract. This attack may only be rectified through intervention and direct modification of the smart contract state by the contract owner.

7 Conclusion

Given the current state of blockchain technology, without fundamentally modifying aspects of the blockchain, there is no perfect interoperability solution. This paper presents a review of three classes of interoperability solutions; time-locks, relaying and notary schemes. Given the trade-offs between interactivity, functionality and centralization we present the TWC; a notary-based generic message passing framework for inter-blockchain communication. As a notary-based framework the TWC trades a decrease in interactivity for an increase in centralization.

The security model provides a fault tolerant interoperability framework where a party may become compromised without compromising the entire system. However, due to the centralization trade-off, once multiple components become compromised an attacker may gain the ability to send malicious messages between blockchains. Even assuming no multi-compromise, bridge operators must still acknowledge the risk of deep chain re-organization, either by malicious actors or a coordinated fork event. Operators must be ready to step in and rectify issues caused by a deep chain re-organization deep in the chain history.

None of the time lock, notary or relay-based frameworks provide the perfect solution for interoperability, however, the TWC presents a starting point for generic inter-blockchain message passing.

References

[1] Vitalik Buterin. Chain interoperability. *R3 Research Paper*, 2016.

[2] Maurice Herlihy. Atomic cross-chain swaps. *arXiv preprint arXiv:1801.09515*, 2018.

[3] Interledger. Hashed-timelock agreements (htlas), 2018.

[4] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. Technical report, Cryptology ePrint Archive, Report 2017/963, 2017. Accessed: 2017-10-03, 2017.

[5] Sergio Demian Lerner. Drivechains, sidechains and hybrid 2-way peg designs, 2016.

[6] Izaak Meckler and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. 2018.

[7] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. ethereum project yellow paper 151 (2014), 2014.

Appendix A Safety Conjecture for Interoperability

We aim to informally illustrate the following:

Safety cannot be guaranteed for an (interoperability) system that establishes forward causality relationships between transactions on two distinct chains, where the chain which is the source for these transactions has probabilistic finality.

A.1 Problem Sketch

- Step I: Build a simple model for a system that establishes a forward causality relationship between transactions on two blockchains (i.e. a bridge)
- Step II: Precisely define a reasonable safety property of such a system.
- Step III: Argue that the safety property of the interoperability system cannot be satisfied due to a dependent safety property of the source blockchain.

A.2 Step I: Problem Modelling

Assume the following construction for a system that establishes a *forward causality* relationship between transactions on two blockchains (referred-to as a "bridge"):

1. Transaction X_1 exists on B_1 at block N_1, sealed at time T.
2. F_1 is the count of blocks elapsed since N_1 at which point X_1 can be considered "sufficiently final".
3. Transaction X_2 exists on B_2.
4. X_2 has a causal relationship to X_1 (i.e. $X_1 \Rightarrow X_2$).
5. X_2 was sealed onto B_2 at block $N_1 + F_1$, at time $T + T_F$ (let T_F = time to produce F_1 blocks on B_1)
6. Block production rates for B_1 and B_2 are roughly similar (impedance matching).
7. Let C_1 be the set of claims that are invariant for blockchain B_1 at block number $N_1 + F_1$. (e.g. account φ has balance of 0 at block number $N_1 + F_1$).
8. Let C_2 be the set of claims that are invariant for blockchain B_2 at some time corresponding to B_1's block number $N_1 + F_1$.

A.3 Step II: Safety Property Definition

Our working definition of a safety property is "[something undesirable] will not happen" . We define the primary safety property of our "bridge" construction:

"For all time after block $N_1 + F_1$, the relationship $X_1 \Rightarrow X_2$ is an invariant in the set of claims $C_1 \cap C_2$"

Another way to state this property is that the safety of our bridge system is violated if there is a non-zero possibility that the causality $X_1 \Rightarrow X_2$ established at time $T + T_F$ may no-longer hold at some point in the future.

A.4 Step III: Proof Sketch

For blockchains with probabilistic-finality in their consensus (e.g. Bitcoin, Ethereum), after a block is "buried under" sufficient proof of work, it is considered "sufficiently final". For probabilistic-finality blockchains, the closed-system assumption[6] is central to the security model; under the closed-system assumption, blockchain reorganizations are considered perfectly safe (by definition of their safety criterion). Although never formally stated, safety criteria for probabilistic-finality blockchains[7], in practice, is understood to be the following:

Assume the set of transactions (T_1, ..., T_n), which successively depend on each other (i.e. for all n, T_n depends on T_{n-1}). Assume these got sealed in the history of the blockchain at block N.

[6] The fact that the blockchain is a self-contained system. All security guarantees provided are reasoned about with this assumption in mind.
[7] Blockchains like Bitcoin and Ethereum

Then, in the course of a re-organization to block N - M such that none of (T_1, ..., T_n) exist in the history, no guarantees are made about the re-inclusion of these transactions in the new history, other than if these transactions are replayed, the transaction will NOT be included out-of-order.

It is perfectly reasonable that in the course of the reorganization, none or some partial contiguous series of transactions get included (starting from T_1) in the new history due to inclusion of other transactions conflicting with one or more of the transactions in the set (T_1, ..., T_n).

By definition of their safety property, these systems can indefinitely tolerate a non-zero probability of a "deep" reorganization event.

Furthermore, it is easy to see from the definition of the safety criteria for a probabilistic-finality blockchain that no event X_1 sealed in a block N_n (n $>$ 0) on chain B_1 is an invariant of the chain. If the event X_1 is the source event in the forward causality relationship $X_1 \Rightarrow X_2$, it follows that this relationship also cannot be invariant (in the intersection of the claims that are invariant for all chains).

Appendix B Sample Operation Using TWC

This section outlines the steps to deploy and use the TWC adapter contracts.

Prerequisites:

1. Deploy Adapter contracts on both Aion and Ethereum. For each contract the following should be set:
   (a) *_signatoryContractAddress*: An address of a contract implementing the *IBridgeSignatory* interface, storing the valid signatory information.
   (b) *_relayer*: An account address that will be used by the bridge to call the Adapter contract to process transfers.
   (c) *_signatoryQuorumSize*: Minimum number of signatures required for a transfer request to be valid.
   (d) *_transactionFee*: Fee associated with performing each cross-chain bridge transaction.
   (e) *_acceptOnlyAuthorizedSenders*: Indicates whether only a pre-authorized set of accounts can request a bridge message transfer.
2. Set the source adapter contract address. Aion adapter will store the Eth adapter address and vice versa.
3. Encoded function in the recipient contract should be accessible by the Adapter contract. Otherwise the function call will fail.

Flow to send a cross-chain message transfer transaction is as follows:

- On the source side of the transfer:
  1. User picks the destination contract, function name and arguments.
  2. A user account or contract calls the *requestTransfer* function with the destination contract address and encoded function call. Optionally the amount of gas that should be used by this function can be set as well. The sender account should be accepted in the adapter contract.
     For example, an encoded function call for
     function setValue(uint128 _v) { value = _v; }
     can be generated using one of the following ways:
     (a) Solidity (v0.4.24):
     abi.encodeWithSelector(bytes4(sha3("setValue(uint128)")), 1)
     (b) Web3:
     web3.eth.abi.encodeFunctionCall(
     {"name": "setValue", "type": "function", "inputs": [{"type": "uint128", "name": "_v"}] }
     , [1]);

  Furthermore, user should include the fee set in the contract with this transaction. If the amount sent is less than *transactionFee*, transaction will be rejected. Otherwise the extra value will be transferred back to the user.

- On the Bridge side:

  If the *requestTransfer* transaction is successful, a *BridgeTransferRequested* event is generated which includes an associated Id (nonce). Bridge will listen for this event and send a request to signatories asking for signatures. Signatories will validate the occurrence of the event and sign the hash of the transfer data. Hash method is Keccak256 on Ethereum and Blake2b256 on Aion.

  Hash (sourceTransactionHash, sourceAdapterAddress, recipientContract, encodedFunctionCall, gas, sourceTransferId, sourceNetworkId)

  Once the required number of signatories have signed the hash, the bridge calls the *processTransfer* function on the destination adapter.

- On the destination side:

  1. The *processTransfer* function is called by the Relayer account through the bridge.
  2. Adapter contract checks the following conditions:
     (a) Transaction hash should not have been processed before.
     (b) Id the of the transfer request should be equal to the expected Id or nonce to maintain order between chains.
     (c) There should be enough signatures included as the input.
     (d) Signatures should be valid
  3. If the conditions pass,
     (a) The recipient contract will be called with the encoded data.
     (b) Transaction hash will be marked as processed.
     (c) An event will be generated indicating the success of the transaction and the status of the function call.
  4. If the transaction hash has been processed before, an event is generated which includes the block number in which the original *processTransfer* transaction was sealed. This can be used to track the original transaction and ensure it's finalized.
  5. In other fail cases, transaction will be reverted.

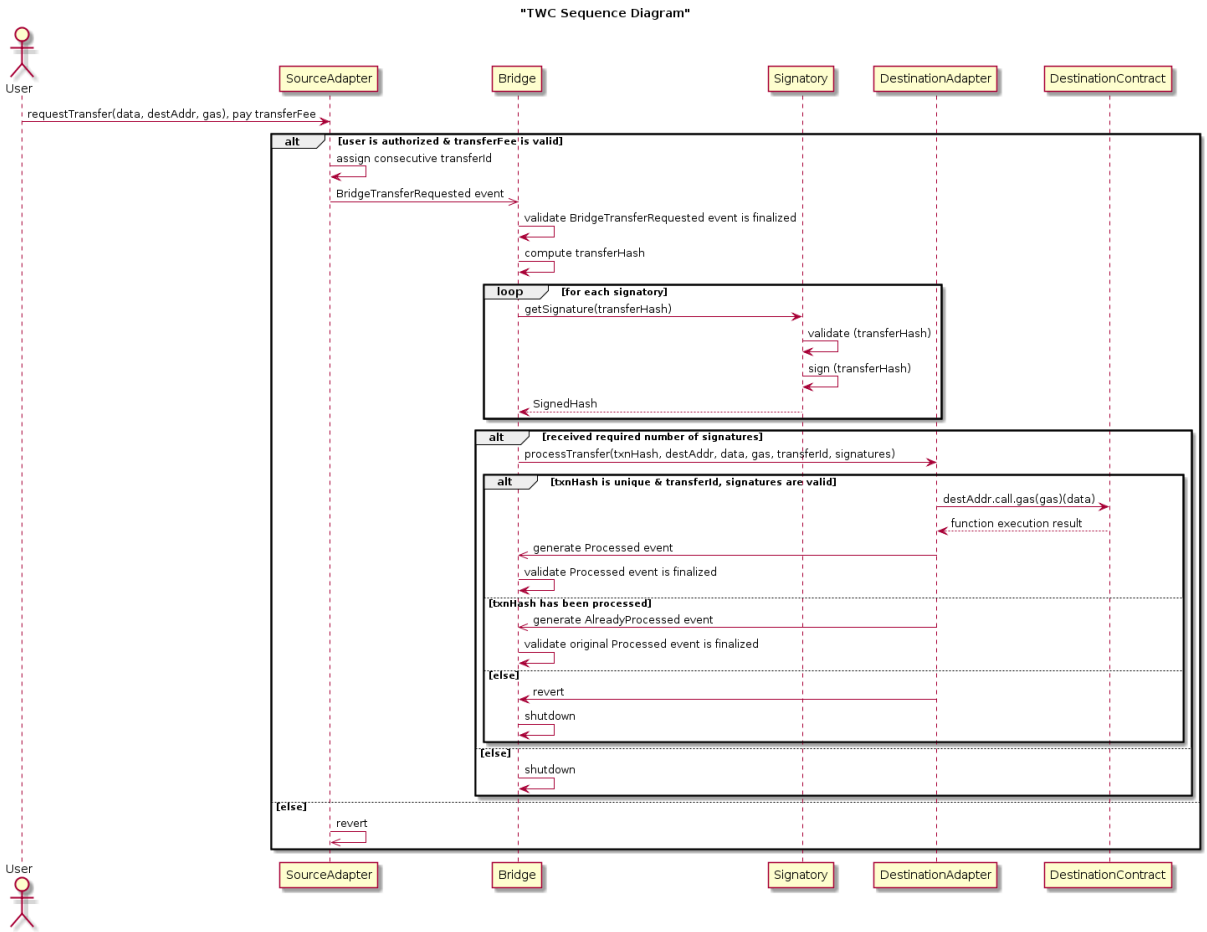

Figure 2: Sequence diagram of the TWC protocol

Appendix C Contract code

We do not recommend using these contracts in a production environment. The code has not been fully tested or audited. Smart contracts for both Aion and Ethereum can be found in `https://github.com/aionnetwork/transwarp_conduit`