

Rigging Specifications

T.R.I.E.

Kris Coward* D.R. Toliver*

with

Cory Sulpizi* Adam Gravitis* Alexander Fertman* Mike Everson*
Robert Moir, Jon Levin

v0.9876, January 2023

1 Introduction

This document serves as a normative specification of *rigging*. A well-formed rig links two things together, such that the integrity of the one is extended to the other. By continuing to build the rig, this integrity-at-a-distance can be maintained between them even if the first is entirely oblivious about the existence of the second.

Rigging is employed a variety of places, including within TODA files, where the proof of provenance is composed of rigs. For a formal analysis of the properties of rigging, see [1]

This document builds up rigging gradually, starting from the smallest components, but we begin with a brief top down overview of the architecture.

A *rig* is a cryptographic data structure that binds an untrusted *leadline* to a trusted *corkline*. The rig provides a simple but powerful guarantee: if the corkline has not equivocated, then the leadline has not equivocated. This holds true regardless of what any other line in the rig has done¹.

Rigs are made by composing *hitches*. A hitch is itself a minimal rig: it provides the rig guarantee over its *topline* and its *footline*. Hitches compose horizontally by *splicing* together and vertically by *lapping* together with existing rigs, allowing them to maintain rig structures under arbitrarily complex circumstances. Checking the validity of a rig is primarily a matter of checking the validity of its hitches and their composition operations.

A *line* is an object that evolves over time. Each update to a line is called a *twist*. Lines are defined by (hash-based) identifiers in their constituent twists, so they have unique histories but potentially equivocal futures.

*TODAQ

¹Note that the rigs described in this work belong to G_{\uparrow} , a supportive guild, as described in [1]

Each twist is composed of *atoms*, the smallest unit of meaning in a rig. In addition to their structural purpose in a rig twists can carry semantic content, which is also expressed as atoms – the subject of the next section.

2 Atoms

The data in a rig is composed of individual atoms. Each atom pairs an identifier (generally a hash) and its corresponding packet. A packet wraps content, adding descriptions of its length and structural shape. This serialization scheme is the Atomic Serialization Protocol.

2.1 Identifiers

An *identifier* is a data structure with the following layout:

1 byte	(hashing) algorithm
n bytes	the output of the algorithm

The current hashing algorithm identifiers:

ID byte	Name	Length	Description
0x00	NULL	0 bytes	Has specialized meaning in rigs
0x22	Symbol	32 bytes	Arbitrary bytes
0x41	SHA-256	32 bytes	The 256 bit SHA-2 digest in FIPS PUB 180-4
0x42	Blake3-256	32 bytes	(not yet implemented)
0x43	Blake3-512	64 bytes	(not yet implemented)
0xFF	UNIT	0 bytes	

The NULL, SYMBOL, and UNIT algorithms all operate without an input packet.

In the NULL and UNIT case, this lack of entropy in the input space is characterized by a zero-length output. Thus NULL and UNIT atoms have one byte in total length (the algorithm byte). By definition nothing hashes to NULL or UNIT (they have no matching preimage)².

The Symbol algorithm 0x22 is always followed by 32 arbitrary bytes. They provide the capacity for meaningless names to be created. Symbols, like the NULL and UNIT hashes, are complete atoms unto themselves³. An identifier whose algorithm is SYMBOL, is called a symbol. Like NULL and UNIT, symbols have no matching preimage, by definition.

Each SHA-256 identifier is characterized by a 32 byte output, hence each identifier using the SHA-256 hash algorithm is 33 bytes in total, with the first byte always being 0x41.

Additional identifier algorithms are expected to be added over time, and implementations must be designed to support the addition of new algorithms.

²Typically the NULL hash is used to signify nothing, either at the start of a sequence or over some duration, and the UNIT hash is used to terminate a sequence.

³Symbols are used whenever a name must be introduced: occasionally within the rigging structure, but much more frequently within the cargo.

Note that algorithm outputs may have lengths other than the 32 bytes of SHA-256.

Notationally, given an identifier I , we employ the following selectors:
 $\text{alg}(h)$ is the algorithm identifier byte of I ,
 $\text{hash}(I)$ is the hash function identified by $\text{alg}(I)$, and
 $\text{digest}(I)$ is the output of $\text{hash}(I)$ applied to the data being hashed, so

$$I = \text{alg}(I) \mid \text{digest}(I)$$

For NULL and UNIT $\text{digest}(I)$ and $\text{hash}(I)$ are undefined. For symbols, $\text{digest}(I)$ is the symbol's 32 bytes, and $\text{hash}(I)$ is the constant function returning those 32 bytes, by definition.

2.2 Packets

A packet is structured content with the following layout:

1 byte	the shape
4 bytes	the content length (big endian)
n bytes	the content

The length slot counts the number of bytes of content, and can therefore accommodate content of up to four gigabytes, after which the data will need to be split across additional packets. This is a 32 bit integer in big endian byte order⁴.

The packet itself is five bytes longer than the length count, because it does not include the shape and length bytes. Zero length packets are not currently allowed.

Notationally, given a packet P , we employ the following selectors:
 $\text{shape}(P)$ is the shape byte of P ,
 $\text{length}(P)$ is the content length, and
 $\text{content}(P)$ is its content, so

$$P = \text{shape}(P) \mid \text{length}(P) \mid \text{content}(P)$$

2.3 Atoms

Atoms are the topmost structure of the Atomic Serialization Protocol, and the fundamental representation of data structures in rigging. An atom is an identifier concatenated with its matching packet.

Given an identifier, its “matching packet” is one where applying the identifier's hash function to the packet yields the identifier's digest value. The whole packet must be hashed, including its shape and length values. Given an identifier I and a packet P , they match if and only if

$$\text{digest}(I) = \text{hash}(I)(P)$$

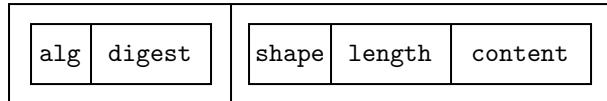
⁴In general, multibyte values used in the atomic serialization format are laid out in big endian byte order, though of course user supplied content (i.e. the contents of arb packets referenced in a cargo trie) may vary.

Notationally, given an atom a , we employ the following selectors:
 $P(a)$ is the packet contained in a ,
 $I(a)$ is the identifier of that packet, and
 $C(a)$ is the content of that packet, so

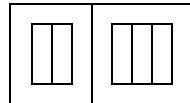
$$a = I(a) | P(a)$$

$$C(a) = \text{content}(P(a))$$

Diagrammatically, we can represent an atom as follows:



Or condensed (we can also partially condense it):



2.4 List of Atoms

Identifiers are self-describing in terms of their length, as are packets. Atoms are therefore also self-describing in terms of their length, and can safely be concatenated one after the other, both at rest and in transit. A list of atoms is also referred to as a *lat*.

Symbols, NULL, and UNIT are complete atoms unto themselves, and are only used in-place within values. They are never included as top level atoms within a lat.

2.5 Errors

None of the error types at this level are fatal (so they should all be treated as having a status colour of "yellow"), but they do differ in their severity. Refer to Section 8.1 for more error handling commentary.

2.5.1 Atomic Errors

There are two cases that result in atomic errors:

- An atom's packet does not match its identifier
- A packet ends before its specified length

These atomic errors may result from a faulty transmission channel, and do not prove anything about the rig. They should generally be considered a case of missing information, not necessarily an error to be reported. The invalid data can be safely discarded, which may trigger a search for the actual atom with that identifier.

2.5.2 Nospec Errors

The following are nospec errors:

- An unknown algorithm byte in the identifier
- A packet with an unrecognized shape byte

Nospec errors occur when the client does not know how to interpret the atom. A nospec error may indicate that a new shape or algorithm is being used that this client has not been updated with, or they may arise from malformed atoms.

2.5.3 Lat Errors

A list of atoms that includes NULL, UNIT, or symbols, is a lat error. A lat error should normally be considered a sign of faulty transmission (though it can also be a sign of the list having been compiled by a non-compliant tool), and is non-fatal (i.e. has a status colour of "yellow").

While a list of atoms containing an atom with an atomic error, is reasonably likely to indicate a problem that warrants discarding much (or even all) of the atoms in the list; we abstain from identifying these cases as lat errors, in order to avoid assigning too many statuses to a single blob of data.

Lat errors are generally fixable by excising the errant atoms. Alternatively, the entire lat could be thrown away and retrieved again.

3 Shapes

The shape of an atom characterizes its structure. This provides constraints for the values that can occupy a given slot, and allows efficient parsing of structured content.

The currently defined shapes:

ID byte	Name	Class	Basic Structure
0x48	basic twist	twist	2 concatenated identifiers
0x49	basic body	body	6 concatenated identifiers
0x63	pairtrie	trie	A map of identifiers, expressed as a list of key-value pairs
0x61	hashes	list	A list of identifiers concatenated together
0x60	arb	arb	Arbitrary binary content

Values are constrained by shape class. For instance, a particular slot might require a twist. Any atom with a shape in the twist class will suffice. The basic twist shape is the first shape in the twist class⁵.

The trie shape class specifically refers to Merkle tries, and all trie shapes carry the guarantee that a specific key has a unique value⁶⁷.

The pairtrie shape is the first shape in the trie shape class. It can be thought of as a Merkle trie with a particularly large branching factor. It is expressed as a lexicographically ordered list of `key` | `value` pairs of identifiers⁸, ordered by key.

The pairtrie content must consist of a strictly positive even number of identifiers, ordered as just described, with no duplicate keys. Any failure to meet these requirements is a shape error (as described in Section 3.1). In particular, it is a shape error for a pairtrie if:

- the pairtrie’s keys are not listed in ascending order,
- the pairtrie’s content contains duplicate keys,
- the pairtrie’s content contains no identifiers (use `NULL` instead),
- the pairtrie’s content contains an odd number of identifiers.

Additional trie shapes are expected to be added over time, and implementations must be designed to support the addition of new trie shapes.

Shapes are used by rigging constructs like twists to specify which atoms can occupy particular slots. Those specifications also include `NULL` as a shape class, which is occupied by exclusively by `NULL`.

Through their specification role, shapes provide the glue between the atom layer and higher layers of rigging.

3.1 Shape Errors

Shape errors occur when a packet provably fails to meet the requirements of its shape, such as a pairtrie that contains only a single identifier.

Shape errors are generally breaking. They represent a provably invalid construction, unlike an atomic or nospec error, which could change if additional information is provided. Only well-formed atoms can have shape errors, and in a well-formed atom the packet uniquely matches the identifier, and is therefore provably erroneous in the cases listed below.

Provably invalid constructions (aka “red errors”) can be useful at higher levels of the system, because given a particular rig its red errors won’t change,

⁵Most shape classes will gradually acquire more instances. In particular, additional body and trie shapes are already on the drawing board.

⁶Or no value at all, when that key can be proven to not exist in that Merkle trie. Except where stated otherwise, no value and the `NULL` value are treated as equivalent.

⁷In order to guarantee that a key has a unique value, it is required that it only appear once in the list; repetition of an entire key-value pair is still a shape error, even though only one value is provided for the key.

⁸e.g. `k0` | `v0` | `k1` | `v1`

whereas errors related to missing data (aka “yellow errors”) may disappear as more information arrives. This may provide opportunities for considering red errors to be a non-operation in cases where it is important to show something could not have happened, providing some amount of resilience to breaking failures of this kind⁹.

The following are shape errors for a given packet:

- A packet of zero length
- Packet contains less than five bytes
- Packet’s shape requires exactly N identifiers but its content does not contain exactly N identifiers
- Packet’s shape requires concatenated identifiers, but the final identifier in its content does not have its required number of bytes (e.g. a SHA256 algorithm byte followed by 10 digest bytes).
- Packet’s shape requires a shape-specific property that can be determined entirely within its atom and the atom does not have that property

4 Twists

Twists are the structure out of which all rigging is created.

The machinery of a twist is primarily concerned with establishing the connections to other twists that are needed to construct a well-formed rig (Section 7).

Both the basic twist shape and the basic body shape specify a fixed list of slots that must be filled by atoms with particular shapes. Inappropriately filled slots result in errors during rig checking.

4.1 Basic Twist Shape

Currently, the only representation of a twist is the basic twist (shape 0x48). The basic twist shape (Section 3) has two slots, which contain identifiers for the following structures:

<code>body</code>	the identifier of the twist’s <i>body</i>	basic body (shape 0x49)
<code>sats</code>	the identifier of its <i>satisfactions trie</i>	trie shape or NULL

Thus a basic twist t has the form

$$C(t) \leftrightarrow I(t.body) | I(t.sats)$$

A twist t also has as its identifier $I(t)$.

To ensure uniqueness of the identifier, the identifier of the body must have the algorithm as the twist’s own identifier. i.e. $\text{alg}(I(t))$ must equal $\text{alg}(I(t.body))$.

⁹The current specification does not take advantage of this opportunity, as it is being considered in the larger context of proof structure resilience.

If the algorithm in the twist's identifier and the algorithm for its body identifier differ then the twist atom was not computed using $\text{alg}(I(t.\text{body}))$, and this is an error. In particular, since it is a failure to meet a structural requirement of the shape, and can be diagnosed without reference to data outside the atom with that shape, it is a shape error.

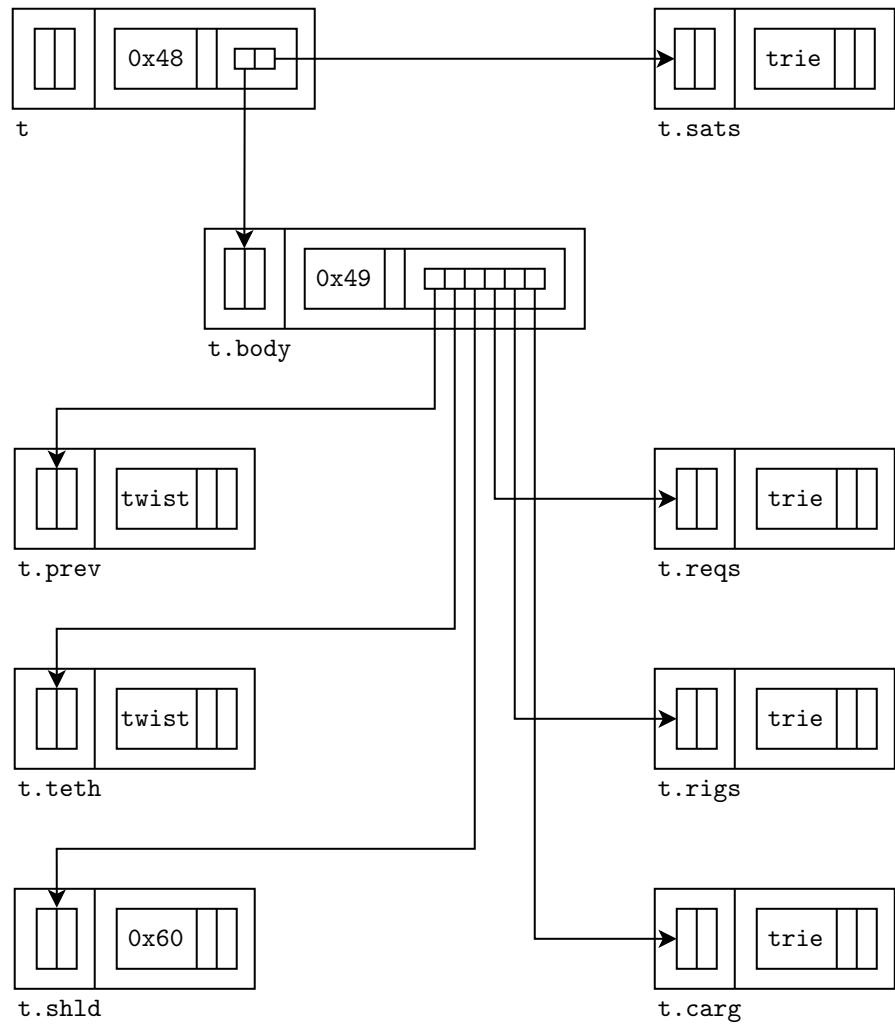
4.2 Basic Body Shape

A twist body must incorporate the following fundamental components into its structure:

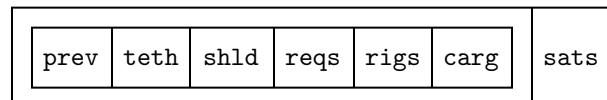
Component	Description	Required Shape
<code>prev</code>	previous twist	twist or NULL
<code>teth</code>	tether	twist, NULL, or UNIT
<code>shld</code>	temporary secret	arb or NULL
<code>reqs</code>	requirements trie	trie or NULL
<code>rigs</code>	rigging trie	trie or NULL
<code>carg</code>	cargo trie	trie or NULL

In the basic twist (shape 0x48), the satisfactions often need to securely reference the rest of the twist data. Aggregating the identifiers of the first six components into a basic body (shape 0x49), provides a body identifier that can be used as this secure reference:

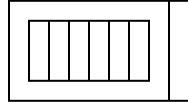
$$C(t.\text{body}) \leftrightarrow I(t.\text{prev})|I(t.\text{teth})|I(t.\text{shld})|I(t.\text{reqs})|I(t.\text{rigs})|I(t.\text{carg})$$



As structures are constructed in which twists are components, the above diagram is too much to reproduce in diagrams of those structures, so twists can be represented in a simplified form (like atoms were above):



Which can be further condensed (or partially condensed):



It is worth noting that since a basic twist body must consist of exactly 6 identifiers, and a basic twist must consist of exactly 2 identifiers; either of these shapes consisting of a different number of identifiers constitutes a shape error.

A twist with a non-NULL tether is called a *fast twist*, and it is made fast by hitching it as described in Section 6. A twist with NULL as the value of its tether slot is a *loose twist*.

The status objects that describe twist errors are found in section 9.2.

5 Lines

Each twist t has a unique previous twist, called its *predecessor*. Continuing this *prev* sequence reveals all the *ancestors* of t . We call such a linked list of ancestors a *line*. Each twist has a single line of ancestors.

In the forward direction a *successor* of a twist t has t as its *prev*, and so on. A sequence of such successive successors are *descendants* of t .

The line of succession of t is not inherently unique, as any new twist may set t as its *prev*. So our first concern is in determining the legitimacy of a claimed successor: does it satisfy the requirements of its predecessor? A twist that does is declared a *legitimate successor*.

In contrast to the notion of a line, which may extend indefinitely into the past and the future, a *segment* has precisely defined endpoints. A segment is a sequence of twists, each a legitimate successor of the last, and it is uniquely defined by its oldest (leftmost) and newest (rightmost) twists. We often use the word line when referring to segments, using the latter term only when disambiguation is necessary.

The status objects that describe line and segment errors are found in section 9.4.

5.1 Requirements and Satisfactions

Each twist may declare *requirements*. If it does, then its legitimate successors must carry *satisfactions* for those requirements. We refer to this system of requirements and satisfactions collectively as *reqsats*.

A twist's requirements are specified in its requirements trie `reqs`, and its satisfactions are provided in its satisfactions trie `sats`. Each entry in `reqs` must be satisfied by a corresponding entry in `sats`. When the `reqs` trie is NULL (and therefore has no keys), it is satisfied by a NULL `sats` trie.

More precisely:

```
function satisfies(x, y)
    if x.reqs == NULL
        if y.sats == NULL
            return true
        return false
    if y.sats == NULL
        return false
    if y.sats.keys != x.reqs.keys
        return false
    foreach (key in y.sats.keys)
        if satisfies(x.reqs[key], y.sats[key]) == false
            return false
    return true
```

With satisfaction defined, we can now also define validity of a twist as satisfying its `prev`'s requirements:

```
function valid(t)
    if t.prev == NULL
        return (t.sats == NULL)
    return satisfies(t.prev.reqs, t.sats)
```

The status objects that describe general requirement and satisfaction errors are found in section 9.3, with errors for more specific reqsat types described in their own sections.

5.1.1 Signature Reqsets

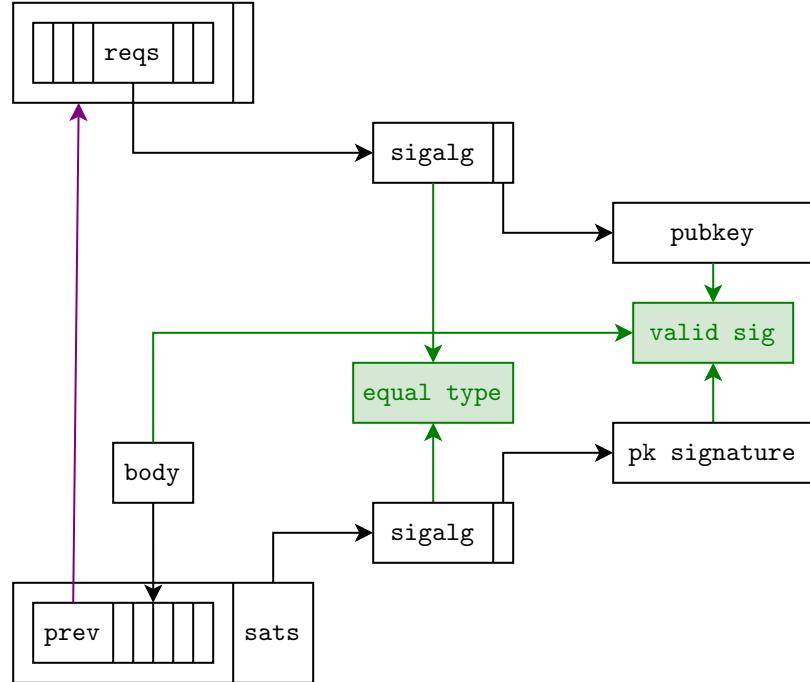
The simplest type of requirement is for a cryptographic signature over the successor using a public key cryptosystem. This provides holders of the private key exclusive control over the creation of legitimate successors.

For a signature reqsat, the key in the `reqs` trie designates a specific type of signature: for example, ECDSA using the `secp256r1` curve and parameters, as implemented in Java.

Let `sigalg` be a identifier representing a type of public key signature, `key.pub` and `key.priv` be representations of the public and private parts of a keypair for that signature type, and `sig(data, key)` be the signature of `data` by (the private part of) `key`; then we incorporate a signature reqsat as follows:

- `x.reqs[sigalg] = I(key.pub)`
- `y.sats[sigalg] = I(sig(body, key))`

The requirement is satisfied if and only if the signature is valid.



The signature algorithms currently available within requirements and satisfactions are:

secp256r1

identifier: 0x22eabd2839f9e57cf2c372e686e5856cf651d7f07d0d396b3699d1d228b5931945
 public key format: Java EC X509EncodedKeySpec (i.e. X.509)
 signature format: Java SHA256withECDSA (i.e. ANSI X.9.62 signature of SHA256 hash) of body identifier (which might not use SHA256)

ed25519

identifier: 0x223d5f4f95cdb1cdfc71014efa1a669fd42599a0ce2000d914a409e48bccaed584
 public key format: Java Ed25519 key (i.e. X.509)
 signature format: Java Ed25519 (i.e. Ed25519 signature per RFC 8032)

5.1.2 List Reqsets

The next type of reqsat is a weighted list of reqsats, referred to as a *rslist*. This allows multiple keys to share control over the succession of an object.

In a requirements trie, the value associated with the rslist key (0x22c9bf129a42fd9478fc42c986ba5b878667) must be the identifier of a list of reqlist entries $I(listEntries[0, \dots, n])$. Additionally, each entry is the identifier of a 2-entry list containing a weight and a requirement: $listEntries[i] = I(weight, reqs)$ where each weight is an arb representing an unsigned 8-bit integer and each reqs is a requirements trie identifier (which is treated the same as a twist's requirements trie identifier).

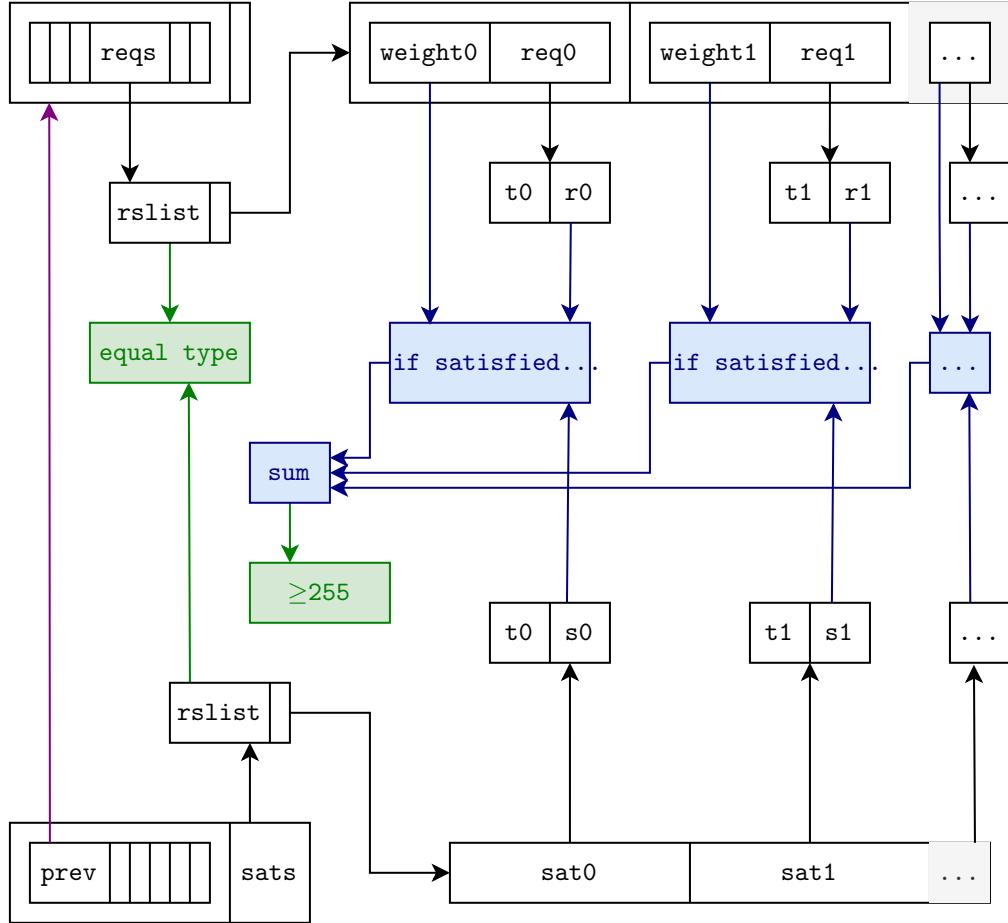
In a satisfactions trie, the value associated with the `rslist` key must be the identifier of a list of identifiers of satisfactions tries (the `satlist`) `I(listSats[0, ..., n])`, where each `listSats[i]` is a satisfactions trie (with the same structure as a twist's satisfactions trie).

To satisfy the `rslist` requirement the length of `listEntries` must equal the length of `listSats`, and the sum of `listEntries[i].weight` across those values of `i` for which `listEntries[i].reqs` is satisfied by `listSats[i]` must be at least 255.

It should be noted that because the `rslist` uses the same format as twists for requirements tries and satisfaction tries, `rslists` can be nested to produce more complex requirements (and satisfactions). Because `rslists`, like all atoms, are referenced using their identifiers (which involve hashes), they form a strict DAG without recursive loops.

It is also worth noting that `NULL` requirements/satisfactions are permitted as members of `rslists`, as they provide functionality, in addition to avoiding special cases for their exclusion.

As an example of this functionality, consider a group of 150 parties who wish to require a $2/3$ majority to validate a successor to a twist. If an `rslist` consisting only of their individual requirements is constructed where they all have a weight of 2, then 128 satisfactions would be needed to validate the successor instead of the desired 100. On the other hand, if their weights are all set to 3, then only 85 of them are needed to validate a successor. However, including a `NULL` requirement with a weight of 155, and assigning a weight of 1 to each of the individual requirements, would produce an `rslist` that controls succession as desired.



6 Hitches

The fundamental unit of rigging is called a *hitch*. A hitch connecting segment A to segment B proves, roughly, that if B hasn't equivocated then A hasn't either.

More precisely, it proves an exclusive relationship between the segment A, the *footline* of the hitch, and the segment B, the *topline* of the hitch. No other hitch can be formed between the oldest twist in A, the *lead* of the hitch, and newest twist in B, the *hoist* of the hitch.

A is therefore *canonical* with respect to B. This provides the mechanism for uniquely selecting a twist's successor. In particular the newest twist in A, the *meet* of the hitch, is the canonical successor of the oldest twist in A, the hitch's lead.

This exclusive relationship is the basis of integrity-at-a-distance, which allows the state of an asset to be managed by an untrusted system, while still maintaining the full integrity of its issuer.

A hitch involves 5 twists:

- the *fastener* `hitch.fastener`
- the *lead* `hitch.lead`
- the *meet* `hitch.meet`
- the *hoist* `hitch.hoist`
- the *post* `hitch.post`

Each position in the hitch has a simple relationship with another, except the hoist, which has a more complicated relationship with several positions.

- `hitch.fastener = hitch.lead.teth`
The hitch's fastener is defined by its lead.
- `hitch.lead = fastPrev(hitch.meet)`
The hitch's lead is the first fast predecessor of its meet.¹⁰
- `hitch.meet = fastPrev(hitch.post)`
The hitch's meet is the first fast predecessor of its post.
- `hitch.post.rigs[hitch.lead] = hitch.hoist`
The hitch's post associates its lead and its hoist.
- `hitch.hoist` is the first successor of `hitch.fastener` to hoist `hitch.lead`

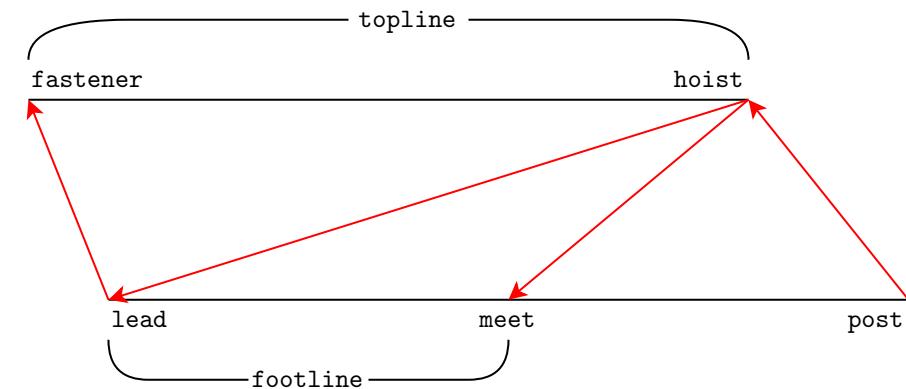
The hitch's *topline* is the segment from fastener to hoist:

`hitch.topline = [hitch.fastener, ..., hitch.hoist]`

The hitch's *footline* is the segment from lead to meet:

`hitch.footline = [hitch.lead, ..., hitch.meet]`

A segment A is *hitched to* a segment B if and only if there is a hitch with `hitch.footline = A` and `hitch.topline = B`.



¹⁰The function `fastPrev` is defined in Section 6.2.

6.1 Hoisting

The heart of a hitch is its hoist. Hoisting a twist binds it to its successor. The hoist of a hitch canonizes `hitch.meet` as the next fast successor of `hitch.lead`¹¹.

The hoist is the one place that coordination is required between the topline and the footline, as information from the footline must be incorporated into the topline. Otherwise the footline can act entirely independently.

Were the hitch to set the key-value pair `[lead, meet]` in its rigging trie, this would be sufficient to link the two from the standpoint of the proof structure.

Operationally, however, anyone who knew the lead could post an alternate value, with deleterious effects on the footline. Additionally, anyone who saw `[lead, meet]` before it was added to the hoist (including the topline operator) could likewise introduce an alternate value.

What is needed is a mechanism that allows values sent to the topline to be accepted *obliviously*. That is, the topline operator should be able to add all received key-value pairs to their rigging trie without introducing the potential for harm. We use the mechanism from [2], which is called *shielding*.

To summarize how shielding is carried out, the footline operator keeps secret the value of the `lead.shld` until the lead gets hitched. This secret value is used in a *shield* function, called S here, which prepends it to an input value, hashes the result into an identifier (using `alg(I(lead))`), and outputs that resulting identifier.

Instead of setting `[lead, meet]` in the hoist's rigging trie, $[S(lead), meet]$ is set. This prevents other parties from trying to occupy that spot before this key-value pair is sent.

A malicious party who sees the new key $S(lead)$ before it is hoisted could still try to fill it with an alternate value. Therefore a second pair of values is required in the same rigging trie; namely, the shielded versions of the key and its value: $[S(S(lead)), S(meet)]$.

Since only the footline operator knows the secret value, only they can construct this second pair correctly. If the value of the first pair is changed, the value in the second pair will not match. The footline operator can easily prove that the second identifier does not match the hash of the first identifier prepended with their secret, which is a condition for this twist to serve as a valid hoist for the lead.

Once the hoist is constructed then the lead's secret value is incorporated into the rig, in order to prove that the meet is its canonical successor. The atom containing the secret value from the lead of a hitch is normally (only) included in a rigging proof when that rigging proof contains the hoist of the same hitch (i.e. once the lead and meet have been hoisted, and the risk of a bogus meet value getting hoisted has been mitigated).

For a more detailed chronology of the generation, use, and disclosure of the secret value. refer to [2]

¹¹Such integrity guarantees are always relative; in this case, to the hitch's topline.

6.1.1 Shield details

We define the *shield function*

```
shield(twist, data) = hash(I(twist))(C(twist.shld) | data)
```

Recall that C takes the content of a packet, so $C(twist.shld)$ is $twist.shld$ without the initial shape and length bytes. Because shields are only used in hitches, and the shield used in a hitch is that of the hitch's lead, we impose the requirement of being an arb shape on $lead.shld$. Setting it to a shape other than arb results in an issue of INVALID in the lead's status.

Note also that because content length is included as a field in the packet of each atom, and the last shapes in the bitstreams (that are hashed to produce shielded values) are all shapes with constant content lengths (defined in this specification), a length extension attack on these shielding hashes will necessarily render that hashed data invalid.

If $twist.shld$ is NULL the above devolves to

```
shield(twist, data) = hash(I(twist))(data).
```

Note that zero-length packets are a shape error, so setting $twist.shld$ to NULL is the only way to achieve this version.

Shields on loose twists are ignored, and should generally be set to NULL.

Given a particular hitch, we define that hitch's shield function $s(hitch)$ as

```
s(hitch)(data) = shield(hitch.lead, data)
```

and require $hitch.hoist.rigs$ to contain the following two key-value pairs:

- $hitch.hoist.rigs[S(hitch, I(hitch.lead))] = I(hitch.meet)$
- $hitch.hoist.rigs[S(hitch, S(hitch, I(hitch.lead)))] = S(hitch, I(hitch.meet))$

As indicated in [2], for the functional value of shielding to be properly realized, the shielded key-value pairs used in the hoist cannot have a matching key and value. While this is of limited concern in the hoist itself (because it requires a collision in a cryptographic hash function), it is relevant to the rest of the topline, which is important to the proof.

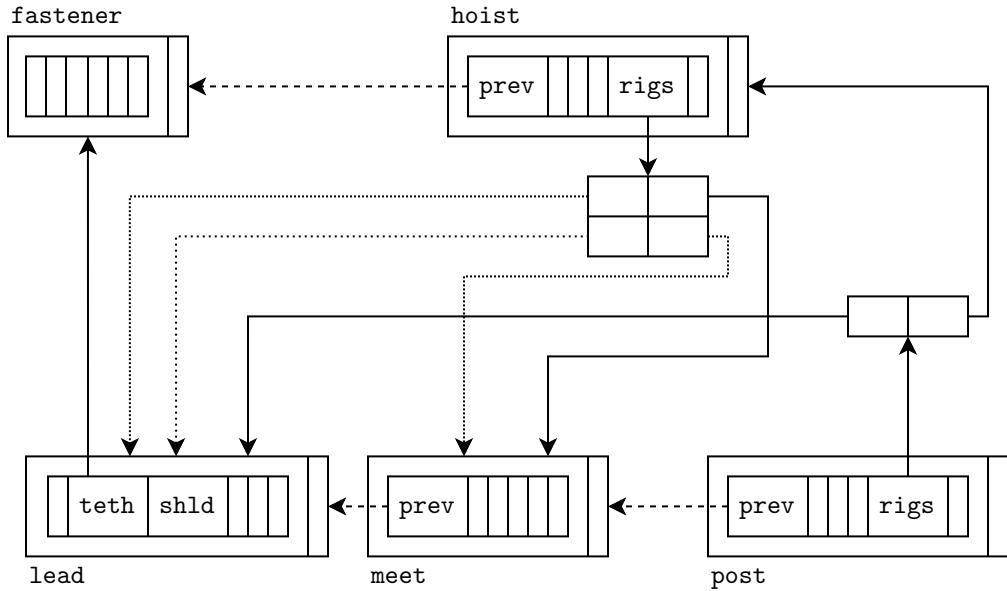
In addition to containing proof that $hitch.hoist$ contains those key-value pairs, the hitch must also provide proof that no twist between $hitch.fastener$ and $hitch.hoist$ contains a pair of entries matching the following description (for arbitrary $data$):

- $twist.rigs[S(hitch, I(hitch.lead))] = data$
- $twist.rigs[S(hitch, S(hitch, I(hitch.lead)))] = S(hitch, data)$
- $twist.rigs[S(hitch, I(hitch.lead))] != S(hitch, I(hitch.lead))$

This proof can consist of proofs of *any* of the following:

- $twist.rigs[S(hitch, I(hitch.lead))]$ is not present in the trie,

- `twist.rigs[S(hitch, S(hitch, I(hitch.lead)))]` is not present in the trie,
- `twist.rigs[S(hitch, I(hitch.lead))]` = `S(hitch, I(hitch.lead))`, OR
- `twist.rigs[S(hitch, S(hitch, I(hitch.lead)))] != S(hitch, twist.rigs[S(hitch, I(hitch.lead))])`



The status objects that describe hitch errors are found in section 9.5.

6.2 Length and Height

A *fast segment*¹² begins and ends with a fast twist (i.e. it has no loose ends). The length of a segment, used in rigging, is defined on fast segments.

To define length formally, we introduce a twist-valued function `fastPrev`, of a twist `t`, which is called the *immediate fast predecessor*:

```
function fastPrev(t)
  if (t == NULL)
    NULL
  else
    if (t.prev.teth != NULL)
      t.prev
    else
      fastPrev(t.prev)
```

¹²In the sense of “fasten”, rather than the sense of having a high velocity.

Less formally, the immediate fast predecessor of t is the closest predecessor to t that is fast (or `NULL` if t has no predecessors that are fast).

With this function and a fast segment L (containing n twists), we can now define the length $L.length$ of our fast segment to be the value satisfying $\text{fastprev}^{L.length}(L[n - 1]) == L[0]$ Or more intuitively $L.length$ is one less than the number of fast twists in L (or the number of sections of fence that can be supported by fenceposts corresponding to the fast twists).

Analogously to a length determining how far away two twists are in time, we also have a height, which determines how far away two lines are in a sort of space, where the unit of distance is a tether. To do this, we introduce a twist-valued function `fastTeth` of a fast twist t , which is called the *fast tether*:

```
fastTeth(t) = if (t.teth.teth != 0)
              t.teth
            else
              fastPrev(t.teth)
```

And thus we define the height of the tether from `twist` to `segment`, $\text{height}(\text{twist}, \text{segment})$ to be the smallest n so that $\text{fastTeth}^n(\text{twist})$ is in `segment`.

7 Rigging

While hitches allow uniqueness of succession in a topline to provide uniqueness of succession in a footline, the tethering must be direct, and the footline's length is limited to 1. In order to allow uniqueness guarantees to be made through indirect tethering, and to provide unique succession for segments with a length greater than 1, we use rigs.

Rigs are assembled from hitches, using rigging operations. A rig R is a data structure which includes a *corkline* ($R.corkline$), whose uniqueness of succession enables the rig to prove unique succession in its *leadline* ($R.leadline$). Much like how in normal fishing operations the corkline being buoyed with cork, makes up the top of the net, and the leadline being weighted with lead, makes up the bottom of the net; in a rig consisting of a single hitch, the topline functions as a corkline, and the footline as a leadline. We also identify a rig as having a height $R.height = \text{height}(R.leadline[0], R.corkline)$.

A rig is constructed from simpler rigs, using the operations we call *splicing* and *lashing*. The simplest rigs (i.e. the inductive/recursive base case) are half-hitches. A half-hitch has the first four twist of a hitch, but omits the post. Each hitch begins as a half-hitch, before its post is added.

Splicing allows a rig to be extended in time, (which in our diagrams is represented horizontally, and may be referred to as the horizontal direction) by combining it with another rig. Lashing allows a tetherline to extend farther from a desired authority which is represented by a corkline (similar to time being horizontal, this is the vertical direction), and does this by combining it with an additional half-hitch.

Because rigs are used to determine whether a particular line has integrity, it is essential that their correct formation be tested. The status objects that shall describe the results of these tests are described in Section 9.6.

7.1 Splicing

To construct rigs across longer lines a composition operation is required that allows a fast successor to be followed by another fast successor. Note that the meet on a half-hitch is a fast twist and therefore has its own tether, which points to the fastener for a hitch in which it is the lead.

We define this operation, called *splicing*, and use it to construct a rig `splicedRig` whose leadline `splicedRig.leadline` has a length `splicedRig.leadline.length > 1`.

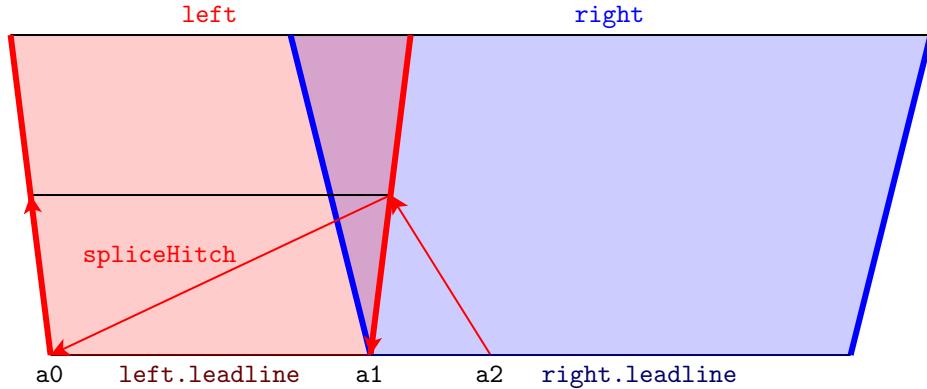
Definition. *If `left` is a rig with the following properties:*

- *its leadline has length `left.leadline.length == 1`*
- *`left.leadline == [a0, ..., a1]`, where `a0` and `a1` are consecutive fast twists, with any number of loose twists between them*
- *there is a hitch `spliceHitch` with `spliceHitch.lead == a0` and `spliceHitch.meet = a1`*

and `right` is a rig with the following properties:

- *`right.leadline.length == n-1 >= 1`*
- *`right.leadline == [a1, ..., an]`, where `an` is fast (`a1` is the same fast twist as specified in `left`)*
- *`a2` (which is possibly equal to `an`) is the fast twist in `right.leadline` with `fastPrev(a2) == a1` (i.e. the first fast twist after `a1`)*
- *`spliceHitch.post == a2` (`spliceHitch` being the same hitch specified in `left`)*

and the corklines `left.corkline` and `right.corkline` of the rigs are aligned then together they constitute a rig `splicedRig` that is said to be obtained by splicing `left` to `right` and whose corkline `splicedRig.corkline` is an enveloping segment for `left.corkline` and `right.corkline`, and whose leadline `splicedRig.leadline = [a0, ..., an]`, which is the enveloping line for `left.leadline` and `right.leadline`.



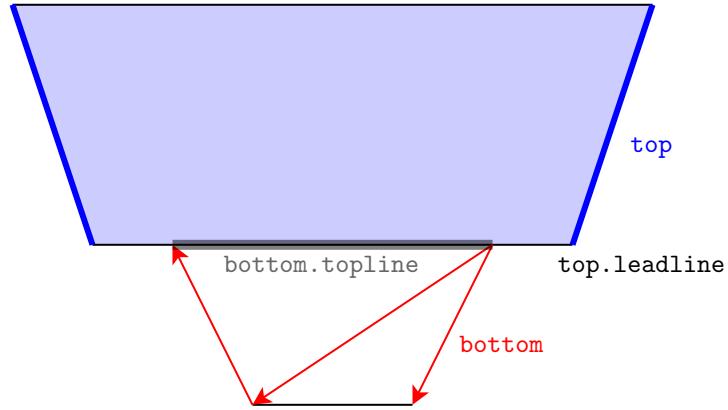
7.2 Lashing

At this point we can now identify arbitrarily many canonical successors along a line, but only if the twists in the line tether directly to the relevant authority. This severely limits the utility of the structures from our guild, so we need to introduce another operation for constructing a rig from simpler rigs. Thus we define the operation of *lashing*.

Definition. *Given:*

- `bottom` is a half-hitch
- `top` is a rig
- `bottom.topline` is a subline of `top.leadline`
- `bottom.fastener` is not present in `top.corkline`

then together `bottom` and `top` constitute a rig `lashedRig`, that is said to be obtained by lashing `bottom` to `top`; with a corkline `lashedRig.corkline = top.corkline`, and a leadline `lashedRig.leadline = bottom.footline`.



The last requirement (that `bottom.fastener` not be present in `top.corkline`) may seem a little odd. It is however, necessary for the proofs of the security guarantees provided in [1].

Roughly speaking, this requirement eliminates the possibility of a corkline tethering (likely indirectly) back into itself¹³, and introducing the risk that one of the times the corkline is reached, it might support a different leadline than on one of the other times. (This risk seems unlikely to be realized, but impossibility is not yet proven.)

Where this is particularly odd is that this requirement is only imposed on the corkline, and not on any intermediate lines (which are entirely skipped over by the requirement). While a tethering loop can be reached before the corkline in verifying a rig, it being a(n infinite) loop will prevent the corkline from ever being reached, and thus prevent a valid rig with that corkline from ever being constructed.

If a tethering loop doesn't prevent the corkline from being reached, then it is at least partly above the corkline, and the portion above the corkline must be ignored. As an example, consider a line `leadline` that is expected to be supported by `corkline0`, where `corkline0` is tethered to `corkline1`, which is tethered to `corkline 2`, which is tethered back to `corkline 0`. If a rig is provided that includes not only the rigging connecting `leadline` to `corkline0`, but additionally includes the rigging connecting `corkline0`, `corkline1`, and `corkline2`, then a rig checker must stop the first time it reaches `corkline0`, instead of proceeding through the tethering loop and declaring the rig invalid on account of the loop's existence.

Thus, a rig checker should return a failure upon detection of a tethering loop, if and only if it occurs before the corkline is reached (if such a failure is not returned, the checker will very likely crash anyway). This failure shall be identified as a `MISMATCH`, and be presented in either a lash or splice status object (depending on how the checker traverses rigs, and which operation it was processing when the loop was detected).

¹³When a line tethers to itself, directly or indirectly, we will call that a tethering loop.

The MISMATCH must only be used if there is a provable tether loop. If an implementation avoids following infinite loops by e.g. setting a height limit on rigs, a red error would be inappropriate, and the failure should be identified in a similar object as UNKNOWN.

7.3 Traversal Algorithm

As rig data is provided as a collection of atoms, it is important to be able to reconstruct the structure of the rig from this collection.

This traversal algorithm determines whether the leadline of a rig is supported by its corkline. It is designed to be applied even in cases where the full collection of atoms is not at hand, and must for instance be queried from a database. As such, it takes as its inputs merely the endpoints of the corkline and leadline, also known as the *corners* of the rig.

The inputs expected by this rig traversal algorithm are:

1. a supporting twist (i.e. corkline end) `z_w`
2. a corkline beginning `z_a`
3. a leadline beginning `f_a`
4. a lat associating identifiers to the atoms/packets they identify

In cases where the full set of atoms is known in advance, and are for instance guaranteed to have no equivocal successors, a simpler rig traversal algorithm may be used. Regardless of which algorithm is used, however, all implementations must construct the same rig structure as is constructed by this example algorithm when given the same rig as input.

The example algorithm relies on a store of lines; this store needs to be able to:

1. record a new line from the rig (`linestore.add(newline)`)
2. extend an existing line from the rig by a new line that overlaps or abuts it (`linestore.extend(line, extension)`)
3. indicate whether a specified twist is a member of a line in the store (`linestore.contains(twist)`)
4. for any twist in the store, identify its successor twist in the line containing it (or indicate that it is the most recent twist in that line) (`linestore.getnext(twist)`)

The algorithm (in pseudocode) follows:

```
function construct_line(start, end)
    line = [end]
    curr = end
    while (curr != start)
        curr = curr.prev
```

```

        line.append(curr)
        linestore.add(line)

        // a function to verify that a line extension is properly supported
        // before updating the linestore with it
        function extend_line(meet, post, hoist, lead)
            if post.rig[lead] = hoist
                extension = [post]
                curr = post
                while (curr != meet)
                    curr = curr.prev
                    extension.append(curr)
                linestore.extend(meet, extension)

        // returns the most recent fast twist that is
        // a predecessor or equal to the argument twist
        function active_fast(twist)
            curr = twist
            while (curr.teth == NULL AND curr != FAIL)
                curr = curr.prev
            return curr

        // returns the first fast twist that is
        // a successor of the argument twist
        function next_fast(twist)
            curr = linestore.getnext(twist)
            while (curr != FAIL)
                if (curr.teth != NULL)
                    return curr
                curr = linestore.getnext(curr)
            return FAIL

        // returns the most recent known fast twist that is
        // a successor of the argument twist
        function last_fast(twist)
            curr = twist
            next = next_fast(curr)
            while (next != FAIL)
                curr = next
                next = next_fast(curr)
            return curr

        // returns the first known successor of fastener with lead as
        // a rigging trie key, FAIL if no such successor is known
        function find_hoist(fastener, lead)
            curr = linestore.getnext(fastener)

```

```

while (curr != FAIL)
    if (curr.rig.keys contains lead)
        return curr
    curr = linestore.getnext(curr)
return FAIL

// returns hoist of hitch for which provided twist is lead
function verify_hitch(lead)
    // follow the lead's tether up to the topline/fastener
    fastener = lead.teth
    // if the topline is already in the linestore, we can try to
    // follow it forward to a hoist
    if (linestore.contains(fastener))
        hoist = find_hoist(fastener, lead)
        // if the linestore line had a valid hoist,
        // we can return it and are done
        if (hoist != FAIL)
            return hoist
        // otherwise we follow the stored line to its last fast twist
        // and identify it as current
        curr = last_fast(fastener)
        // and set the current twist's fast predecessor as
        // the previous fast twist
        prev = active_fast(curr.prev)
        // if the topline isn't already in the linestore,
        // we need to move up another level to be able to move forward
    else
        // we begin by identifying the lead for the hitch
        // that our fastener is in the footline of
        toplead = active_fast(fastener)
        // and then we use that lead to fill in the hitch
        tophoist = verify_hitch(topleft)
        // with the hitch filled in, we can set its meet as our current twist
        curr = tophoist.rig[topleft]
        // and load this new footline into the linestore
        construct_line(topleft, curr)
        // we check the new footline for the hoist of our original footline
        hoist = find_hoist(topleft, curr)
        // and if it is present, we can return it and are done
        if (hoist != FAIL)
            return hoist
        // otherwise its meet is already set as the current twist,
        // so we set its lead as our previous to continue
        prev = toplead
        // having followed the topline for at least one fast twist
        // and failed to find a hoist, we continue to extend it forwards,

```

```

// one footline at a time, until we find a hoist
while (hoist == FAIL)
    // we set up hoist information to verify proper support
    // of the new segment we're constructing
    // note that this code requires that tophoist is
    // the hoist supporting the last segment of the stored line
    // (the provided pseudocode does not make this the case
    // when extending a stored line)
    oldhoist = tophoist
    tophoist = verify_hitch(curr)
    next = tophoist.rig[curr]
    extend_line(curr, next, oldhoist, prev)
    // now that the curr-next segment is in the linestore,
    // we check it for the hoist we were originally looking for
    hoist = find_hoist(curr, lead)
    // and advance our previous and current twists, in case it wasn't there
    prev = curr
    curr = next
    // if the hoist was present, the while loop will exit,
    // and we can return it;
    // otherwise we try moving forward again
    return hoist

// returns last successor of f_a that is supported by z_w
function main(f_a, z_a, z_w)
    // set up our corkline
    construct_line(z_a, z_w)
    // find the hoist of the hitch that supports
    // the next fast twist in our leadline
    hoist = verify_hitch(f_a)
    // store the footline of that hitch and
    // get ready to extend it forward
    curr = f_a
    next = hoist.rig[f_a]
    construct_line(f_a, next)
    // keep extending forward as long as we can
    while (hoist != FAIL)
        oldhoist = hoist
        prev = curr
        curr = next
        hoist = verify_hitch(curr)
        if (hoist != FAIL)
            next = hoist.rig[curr]
            extend_line(curr, next, oldhoist, prev)
    // when we can no longer extend forward,
    // return the last twist we reached

```

```
    return curr
```

For an example of the steps of this algorithm applied to a rig (with several irregular features which might not be traversed correctly by a poorly implemented traversal function) refer to Appendix B.

8 Commentary

8.1 Error Handling Notes

An invalid atom, for instance one with a packet length that is longer than its content, can cause subsequent atoms to be considered invalid as well, since their offsets will be misaligned. Ensure you check the validity of atoms before you write them.¹⁴

Up-to-date clients should treat atoms with nospec errors carefully: unlike atomic errors, they are not recoverable by simply re-requesting the information, but unlike shape errors they are not provably broken. Storing them for re-examination after updating the client could result in an exploitable memory leak. They should be discarded and ignored, and proofs that contain them considered broken for most purposes: just not provably broken.

8.2 Evolution and Modularity

Rigging is built using cryptographic primitives which are the subject of ongoing cryptanalysis. Between this cryptanalysis and more general advances in computing, these primitives can be expected over time to lose the properties necessary to support security guarantees. Consequently, it is helpful to point out which parts of this protocol are expected to remain static over time, and which are expected to evolve, so that this information can inform implementation decisions (e.g. around modularity).

The fundamental nature of shapes, packets, identifiers, and atoms is expected to remain static, however new hash algorithms and new shapes are expected to be introduced. Some of these shapes may also be introduced into new class (which may also incorporate existing shapes; e.g. if a novel representation of a twist body is developed, then "twist body" would become a new class of rigging shape, which would incorporate the existing "basic body" shape).

The fundamental role and essential constituents of a twist are expected to remain static (even if new representations emerge), as are previous, next, successor, and predecessor relations, the definition of a line, and the role of a reqsat. The *role* of a reqsat remaining static however, does not preclude the introduction of new types of reqsat (signature type reqsats being an obvious example of a type into which new replacements will need to be introduced, as cryptographic technology progresses, and old signing algorithms cease to be secure).

¹⁴Particularly if they are being rebroadcast.

All of the rigging structures and operations introduced so far (i.e. hitches, rigs, lashing, and splicing) are not expected to change significantly, though the preconditions on the operations (e.g. that no twist in the bottom half-hitch in a lash be present in the corkline of the top rig’s corkline) may be relaxed or removed as they are proven unnecessary for the security guarantees a rig provides. Additionally, new operations are expected to be introduced to allow the construction of more diverse rigs (e.g. rigs where the footline of a hitch can support the topline, and these rigs can be combined with each other and with existing rigs). No changes to rigging will be made however, which undermine the fundamental rigging guarantee: that any two twists `succA` and `succB` which are supported as successors of the same twist `init` by colinear supports `suppA` and `suppB` will be colinear themselves.

The general model used for verification (i.e. the status objects and their nesting structure) will also remain static, however evolution in other parts of the protocol will have corresponding evolutions in the status objects. (E.g. the introduction of a new type of `reqsat` will be accompanied by the introduction of status objects for that `reqsat` type; or the relaxation of preconditions on a rigging operation will be accompanied by the removal or reduction of scope of red statuses for the relaxed condition being `unmet`).

9 Status Abjects

When presented a list of atoms that do not form the expected rig, it can be difficult to determine what went wrong through manual inspection. Status abjects (using the abjects defined in [3]) provide a way for that information to be communicated in an interoperable form, similar to a stack trace.

9.1 Description

9.1.1 Basics

For each defined structure within a rig, whether it is a concrete structure represented exactly by an atom (e.g. a twist body), or a logical structure composed of multiple concrete structures and the relations between them (e.g. a half-hitch or a `reqsat`), there is a structure status, which is presented as a DI.

In order to justify a result, it is often necessary for a structure status to refer to the status of subordinate structures, whose status structures are made available as children of the initial status structure. These relationships are described in 9.1.4.

The status abject for the rig then consists of the structure status for the rig itself (which is the focus of the abject). If the rig’s structure status (or any other included structure status) has a non-green colour that is determined by its children, then the structure statuses for those children shall also be included.

A structure status DI has the following fields:

<code>structype</code>	SYMBOL	indicates the type of structure being checked
<code>colour</code>	SYMBOL	indicates the status of the structure (good, bad, or unknown)
<code>issue</code>	SYMBOL (OPTIONAL)	indicates the problem with the structure (if there is one)
<code>children</code>	TRIE (OPTIONAL)	children of this status abject, as described in Section 9.1.4
<code>reference</code>	UNDEFINED (OPTIONAL)	the hash of the specific instance of the structure, whose status is described by this DI

The symbols representing these fields are as follows:

- `structype`
0x2299c4857f9cc846feed9155d2df4f25f79f15ecfedaa3ac6ddfa3d102a3c503af
- `colour`
0x2266fccf371f70335ab69bb4157f4a06f3588defddd4493c2490304dc30b80ce18
- `issue`
0x22027072c8fa69e32fa6cb4d6adcb8d0d5182a186bfb21d817ffb3039bc6c286
- `children`
0x22bc2d4a3c033b26b15eaa8aafe96fd95370e26779aba66c05b13335ade086ce27
- `reference`
0x2242cb1cadebe0f6d30a7352322580225e50fc1730e222eb5c8eaac068d63f02cc

9.1.2 Colours

The colour of the focus structure status indicates whether the provided data constitutes a valid rig (green), might constitute a valid rig (yellow), or cannot constitute a valid rig (red). In descendant structure statuses, the colour indicates whether an object is provably good, does not have enough information to prove whether it is good or bad, or is provably bad.

Thus the colours (and the symbols used to represent them) are:

- `green`: proven to be good 0x2224691d8ddb740e9909846dac2cd1356b3328b15583fde539f94b250262edf6
- `yellow`: cannot be proven to be good or bad with the information provided
0x224642657982312fd36471bfb468b650fbc1cad3c571c079178109f709a41c8a5
- `red`: proven to be irredeemably bad 0x22ed860e74ea0574b0fd5a275804c00f68ce73a1aa06e22c1ab322f

The colour of structure status is determined according to the following rules:

1. The status is marked as `green` if the result is not marked with any other colour, and does not inherit a colour from any of its children
2. The status is marked as `red` if the result is marked with a `red` issue (ie. `INVALID`, `MISMATCH`)

3. The status is marked as **yellow** if the result is marked with a **yellow** issue (ie. **MISSING**, **UNKNOWN**) and is not marked with a **red** issue
4. The status inherits the colour of its children using the following rules, if it is not marked with an issue providing colour of its own:
 - (a) If the result has been assigned a colour due to any described “Colour Rules”, the result’s colour does not change
 - (b) Else if any of the result’s children are marked as ‘red’, then the result itself is marked as ‘red’
 - (c) Else if any of the result’s children are marked as ‘yellow’, then the result itself is marked as ‘yellow’
5. A rig checker may treat an atomic error as a **yellow** error, and/or a shape error as a **red**, however it may also terminate without providing a status at all on either of these errors.

9.1.3 Issues

Each structure status has a list of potential issues. The order of the list is important: the first issue encountered will be the issue assigned to that status object.

Issues are assigned one of the following types:

- **MISSING**: indicates that a piece of required information was not provided
- **UNKNOWN**: indicates that it was unclear how to interpret a piece of information
- **INVALID**: indicates that the provided information or satisfaction is invalid (e.g., the given satisfaction does not satisfy the given requirement, a given packet has the incorrect shape, etc.)
- **MISMATCH**: indicates that two provided pieces of information do not align

Additionally, atomic errors and shape errors may also be coded as issues, and are given symbols for this purpose. When this is done, these errors take precedence over any status (as they should, since they often render a status impossible to compute).

The symbols used to represent the issues are:

- **MISSING**: 0x22dad0e4311d00c017ea5f8f8304b99bbbc93d798d3d7c0d1af4d0aaae164e843b
- **UNKNOWN**: 0x222a0ed5cbc7b198a3d088f2c70c6681c8164a90fdd74d9496171b25e08b1eaf78
- **INVALID**: 0x22c9f630e9dbc8d55bf4d0bebb6051b5b587863f31fe734404bedfad012608a4
- **MISMATCH**: 0x2293ca80267f8eed7ed9d8c8b6a4755c87f91e12d1e52ce0f8a1b946d2b78a7c1b
- **atomic error**: 0x2223fd016f0a8653cca6e7b679fa69e579ce5fc76443a8c75a7a18551a63d647b8
- **shape error**: 0x220ad9dd65d2d069c61f8cb9b2c449c3cee145b7f459537feeb8f4b2b4b2defdfe

9.1.4 Children

Each structure status description includes a list of structure status types which may be present as keys in the `children` trie. Sometimes there are multiple potential children of the same type; in these cases keys are provided in the description of the parent structure status type, and these keys are used instead of the child's type. When a listed status type key is present, the associated value is the identifier of the DI implementing that structure status, which provides the status of the corresponding child object. Any `green` children are omitted from the 'children' trie.

Additionally, some children are given identifiers that end in `[]`. As is convention, this denotes that the child in question is actually a list. Because the children trie is simply a trie, and not a DI with a schema, this list is provided without any abject machinery. The value associated with a key whose identifier has `[]` appended is the identifier for an atom with the `0x61` "hashes" shape. The identifiers for the child DIs from the lists are then included in that shape in the order described in the child description.

The 'children' trie of a status object should be omitted entirely if:

- The status object itself is `green`
- The status object itself has an issue
- There are no non-`green` children to include

9.1.5 Logical Objects

Logical objects do not have a naturally corresponding atom: for example, a `reqsat` is a combination of a requirement object and its children with a satisfaction object and its children. Lacking a natural atom from which to draw their identity, they wander the nomenclature wastelands, seeking the solace of an alternative reference wherever it may be found.

9.2 Twist Structure Statuses

These are the results from the checks to ensure that provided twists are consistent with the description in Section 4.

```
twist
0x221267ca19a66c372b90849cb4f807f6c22b1f36c1057f6bba4f5209b12827beae

Possible Issues
MISSING: Missing the packet
INVALID: Not a twist

Children
twist-body
```

twist-body

0x2295085430dc7c03412d07e4ff91dd5c2c961cccd6ef910f63a408d6f315e41fa3

Possible Issues

MISSING: there is no atom available whose identifier matches the body identifier
INVALID: the shape of the body packet is not a twist body shape

9.3 Reqsat Structure Statuses

These are the results from the checks to ensure that provided requirements are satisfied in provided satisfactions in accordance with the description in Section 5.1.

reqsat

0x07b12223bd3d6ce73ed9f70813e7a00d9621326a288540ea53e3a9bf1ce0857e

Notes

- This is a logical object; there is no “reqsat” atom in the rigging. It has no reference value.
- If the ‘reqtrie’ child or the ‘sattrie’ child are non-‘green’, the other children are not calculated and therefore should be omitted.

Possible Issues

MISMATCH The requirements trie is NULL XOR the satisfactions trie is NULL
MISMATCH The keys in the reqtrie differ from the keys in the sattrie
UNKNOWN A key in the reqtrie and sattrie does not correspond to a known reqsat interpreter

Children

reqtrie	A reqtrie structure status
sattrie	A sattrie structure status
rslist-reqsat	Only present if both tries contain an rslist
secp256r1-reqsat	Only present if both tries contain a secp256r1 signature
ed25519-reqsat	Only present if both tries contain an ed25519 signature

reqtrie

0x22298f51b6a027ee01e7cc628818018c9e5361aa15ecb2ead84c4c40f15344f8f4

Possible Issues

MISSING The atom corresponding to the reqtrie identifier was not provided
INVALID The provided reqtrie is neither NULL nor a trie

No Children

sattrie

0x22f05766cc59b6d87d8b81020058b67900bf8195418ba87abe851ee22284da9dfe

Possible Issues

MISSING The atom corresponding to the sattrie identifier was not provided

INVALID The provided sattrie is neither NULL nor a trie

No Children

9.3.1 List Reqsat Structure Statuses

rslist-reqsat

0x22c9bf129a42fd9478fc42c986ba5b8786675ee42109cd3a9fdb208f4e9654148

Notes

- This is a logical object; there is no “rslist-reqsat” object in the rigging.
- This object does NOT inherit the colours of its ‘rslist-req-entry’ children.
- If the ‘rslist-req’ child or the ‘rslist-sat’ child are non-‘green’, the ‘rslist-req-entries’ child is not calculated and therefore should be omitted.

Special Rules The **rslist-reqsat** object shall be coloured **green** if and only if The total sum of all ‘green’ rslist-req-entries is ≥ 255 .

The **rslist-reqsat** object shall be coloured **yellow** if it shall not be coloured **green**, and

- The sum of weights of all **green** and **yellow** rslist-req-entries is ≥ 255 , or;
- A **rslist-req-entry** child with a **yellow** **rslist-req-weight** child is **non-red**

Otherwise, the **rslist-reqsat** object shall be coloured as **red**.

Possible Issues

MISMATCH The number of requirements differs from the number of satisfactions

Children

rslist-req

rslist-sat

rslist-req-entries A list of status objects, corresponding to the entries defined in the requirements list. Note that this child is only returned if both other children are green.

rslist-req

0x2241c5142f5ac4cd48cdadd6c11f3c17106b4c7c31b160c241f36b6f65aca45e8e

Possible Issues

MISSING The atom corresponding to the rslist-req identifier was not provided

INVALID The packet corresponding to the rslist-req identifier does not have hashes as its shape

No Children

rslist-sat
0x2232088b6ba2cfb1a8116b096e5af5eadb1986b2b1b9e168cb17779f2ee48277ea

Possible Issues

- MISSING The atom corresponding to the rslist-sat identifier was not provided
- INVALID The packet corresponding to the rslist-sat identifier does not have hashes as its shape

No Children

rslist-req-entry
0x22362e4554299e2ad4e20a13712ada3c23175d129ae0741abdfc2bde7d4165e706

Possible Issues

- MISSING The packet corresponding to this entry was not provided
- INVALID The packet corresponding to this entry is not a hashes packet

Children

rslist-req-weight
reqsat

rslist-req-weight

0x2227f2fb1e21f73fc5d25f314eeaaa3a693d4083bda1b6cc01135b9d5b54f6517

Possible Issues

- MISSING The packet corresponding to this weight was not provided
- INVALID The packet corresponding to this weight is not an arbitrary packet with a 1-byte content

No Children

9.3.2 Signature Reqsat Structure Statuses

secp256r1-reqsat
0x22eabd2839f9e57cf2c372e686e5856cf651d7f07d0d396b3699d1d228b5931945

Notes

This is a logical object; there is no secp256r1-reqsat atom in the rigging. It has no reference.

Possible Issues

- INVALID the provided signature is not valid

Children

secp256r1-req
secp256r1-sat

secp256r1-req

0x2210f601d3262625b2f7ac3d51110f61333755b66d4433261b148aaddeaffd75cec

Possible Issues

- MISSING The packet corresponding to this req was not provided
- INVALID The packet corresponding to this req is not an arbitrary packet

No Children

secp256r1-sat

0x22fd10613c50d9199489e8f56c723dc09c3cefec00e55a7e93a2fb3f46a1a5f76a

Possible Issues

MISSING The packet corresponding to this sat was not provided

INVALID The packet corresponding to this sat is not an arbitrary packet

No Children

ed25519-reqsat

0x223d5f4f95cdb1cdfc71014efa1a669fd42599a0ce2000d914a409e48bccaed584

Notes This is a logical object; there is no “ed25519-reqsat” object in the rigging. This structure status has no reference.

Possible Issues

INVALID the provided signature is not valid

Children

ed25519-req

ed25519-sat

ed25519-req

0x2261443389b6e2999cd2c25c054be8a11657fb6a06820943bc2cf9229d424c1a12

Possible Issues

MISSING The packet corresponding to this req was not provided

INVALID The packet corresponding to this req is not an arbitrary packet

No Children

ed25519-sat

0x22b2cfec6188557738125b7e4d41dfd3bd82e38e453aecb747ae9d741d129fa296

Possible Issues

MISSING The packet corresponding to this sat was not provided

INVALID The packet corresponding to this sat is not an arbitrary packet

No Children

9.4 Line Structure Statuses

These are the results from the checks to ensure that provided lines are consistent with the description in Section 5.

succession

0x227fd8ded12940e8b4d16001ab8bc1b80c55052eadba8bd707a6df766d5f50e3ef

Notes A succession is a logical object. The ‘successor’’s twist is used as a reference.

Children

successor
predecessor
reqsat

A **twist** status object corresponding to the successor
A **twist** status object corresponding to the predecessor
the **reqsat** for the relation in question (req from predecessor, sat from successor)

line-segment

0x22366a8aa7e49d39bd21eb95cf26c6b6e899d9213b4c80b638c7d352b68c24954e

Notes This is a logical object. Its reference is the identifier for the most recent (newest) twist in the line-segment. Given a fast twist **start**, we walk backwards until we reach another fast twist

Possible Issues

INVALID the **start** twist is not a fast twist (either a loose twist or a non-twist shape)
UNKNOWN the **start** twist has a shape that has not (yet) been assigned
MISMATCH NULL is reached

Children

twist
succession[]

the twist result for the **start** twist
A reverse-chronological list of succession objects, starting at a fast twist and ending when it reaches another fast twist. If a succession object is non-green, this list is short-circuited

9.5 Hitch Structure Statuses

These are the results from the checks to ensure that provided hitches are consistent with the description in Section 5.

hitch

0x22d0eb6e7af1ca49d58104b3862955078f2e33a1c67645e15459d6808157fd5c88

Note

- A hitch is a logical object; its reference is the identifier of its post
- The half-hitch (child) structure status may be returned with an invalid lead-footline before the post-key is calculated

Possible Issues

MISMATCH The terminal twist of the post-footline does not match the meet of the half-hitch, or the post-footline contains a fast twist other than at its endpoints

Children

post
post-footline
post-key
half-hitch

A **line-segment** object starting at the post
The **half-hitch** object.

half-hitch

0x222008d2f5c875b46e4f52896ecae0c12f686cc43f57409bde83760fe50388b868

Notes

- The half-hitch is a logical object; its reference is the identifier of its meet
- The half-hitch status may be shortcut in the event that the hitch cannot locate the lead (e.g. the **lead-footline** is invalid). In that event, the **lead-footline** will be the only child returned.
- The **fastener** is intentionally omitted as a child; it is a grandchild via the **topline**

Possible Issues

MISMATCH The terminal twist of the lead-footline does not match the lead

Children

lead

hoist

meet

lead-footline A line-segment object starting at the meet

topline

lead

0x2288dc9ca034029a13c404ecf210d42698c9971ad78594d72c6e790dc5c4945b2f

Notes

Because the lead is a twist, its reference is the identifier for the twist which instantiates it.

Possible issues

INVALID The lead's tether has a known non-twist shape, or its shield is something other than an arb or NULL

MISMATCH The hitch being checked has a specified fastener, and the lead's tether is a twist other than that fastener

UNKNOWN The lead's tether has an unrecognized shape

MISSING There is no provided atom whose identifier matches the lead's tether

Children

twist

hoist

0x22863bfd943c67ed29ee7a3e199ac281e93df674809a4a7795317ded106210b30

Notes

Because the hoist is a twist, its reference is the identifier for the twist which instantiates it.

Possible issues

MISSING	there is insufficient trie data to establish whether the provided lead is a key ¹⁵
INVALID	the hoist's rigging packet has a non-trie shape
UNKNOWN	the hoist's rigging packet has an unassigned shape
MISMATCH	the hoist's rigging trie is provably incapable of hoisting the provided lead (i.e. there is a null proof for <code>shield(lead, I(lead))</code> or for <code>shield(lead, (shield(lead, I(lead))))</code> ; or <code>rig[shield(lead, (shield(lead, I(lead))))] != shield(lead, (rig[shield(lead, I(lead))]))</code> ; or <code>rig[shield(lead, I(lead))] = shield(lead, I(lead))</code>)

Children

`twist`

`meet`

0x22a94d1bc7016fb4422f0dab55c9c3800b9b98d70cbbb4198a8bc4bff7c15c9bb8

Notes

Because the meet is a twist, its reference is the identifier for the twist which instantiates it.

Possible issues

INVALID The meet's tether is NULL

Children

`twist`

`post`

0x224abe314aa6feb47cdfdc2b07c00c64a2b668f0d9edd4454d95e6c8c01254e73c

Notes

Because the post is a twist, its reference is the identifier for the twist which instantiates it.

Possible issues

UNKNOWN The post's tether has an unassigned shape

INVALID The post's tether is NULL or has a non-twist, non-UNIT shape

Children

`twist`

`post-key`

0x227290df7d067bf0608a4785024376d744e50b7c4d285ddd5c3b932e8e78e3ce26

Notes

This is treated as a logical object. Its reference is the twist identifier for the post whose rigging trie is being examined (i.e. the same reference as the sibling post structure status).

Possible Issues

- MISSING** there is no atom whose identifier matches post.rigs
- UNKNOWN** the post's rigging has an unassigned shape
- INVALID** the post's rigging has a non-trie shape
- MISMATCH** cannot find the 'lead' in the post's rigging, or the value for that key is not the hoist's identifier

No Children

topline

0x22031c1b1fc93d62b5cac2456f2408a0e4fe6c30560efc82a4270864fbb779e520

Notes

This is a logical object. Its reference is the hoist for the hitch it is the topline of (which is also its most recent twist).

Possible issues

- MISMATCH** the provided topline does not include the provided fastener as one of its twists (i.e., NULL is reached)

Children

succession[]

A reverse-chronological list of succession objects, starting at the hoist and ending at the successor of the fastener. If a succession object is non-green, this list is short-circuited

topline-key[]

A reverse-chronological list of topline-key structure statuses, starting at the predecessor of the hoist and ending at the successor of the fastener.

topline-key

0x22e335eaae9e2cbbf1f1139999dd67c7ad9370a63d30e16a6eb4615d158b459e7b

Notes

This is treated as a logical object. Its reference is the identifier for the twist (in the topline) whose trie contents are being reported on.

Possible issues

- INVALID** the rigging shape is a known non-trie, non-NUL shape
- MISMATCH** the rigging trie provided satisfies the hoist condition for the provided lead
- MISSING** there is insufficient rigging trie data to establish whether the trie satisfies the hoist condition
- UNKNOWN** the rigging shape is not a known shape

No Children

9.6 Rigging Structure Statuses

These are the results from the checks to ensure that provided rigging is consistent with the description in Section 7.

Because each rig could be either of: a single half-hitch, the result of a lashing operation, or the result of a splicing operation, the `rig` status object is taken to include all of these things. In particular, one of the operands in the splicing operation (the one whose leadline has a length of 1) must be either a single half-hitch, or the result of a lashing operation. Likewise, one of the operands in the lashing operation must be a single half-hitch.

Thus our `rig` object is taken to have a `rig` and a `splice` as optional children, which are null if the rig's leadline has a length of 1. The rig supporting the most recent fast segment of the leadline is handled in the remaining objects.

Of these remaining objects, there is another `rig` object, and a `lashing` object, which are also optional (they are null if the most recent fast segment of the leadline is supported by a single half-hitch), and a mandatory `half-hitch` object, which always represents the half-hitch whose footline is the most recent fast segment of the leadline.

`rig`

0x22d7357365f862a15d5fc6de991334e96c084bbabfcff1150425c9944fb3937a9e

Notes

- The reference for this structure status is the identifier of the earliest twist on its leadline.

Possible Issues

INVALID The corkline does not support this rig's leadline

MISMATCH The leadline of this rig is not disjoint from its corkline

Children

<code>corkline</code>	The corkline specified by arguments to the check
<code>lash</code>	
<code>splice</code>	This object is only present if this rig is being spliced onto a previous rig
<code>half-hitch</code>	The half-hitch whose lead is the earliest twist on the rig's leadline
<code>hitch</code>	A hitch object with a post equal to the meet of the half-hitch specified above. This object is only present if there is a splice

`corkline`

0x222db7e19336b76c355ec281232f9aad367d4a8544f092e4f147ca5bf7344cd8e

Notes

This is a logical object. Its reference is the identifier for the most recent twist in the corkline.

Possible Issues

MISMATCH The corkline traversal reached NULL rather than the specified earliest twist

Children
succession[] A list of succession objects in reverse chronological order, starting at the specified end twist

splice
0x22fd2e95961c3986ff20ea8371b2406f24fe893a1faaf7759276c9704c1143aa3c

Notes

This is a logical object. Its reference is the identifier of the twist in the headline that is common to the two rigs being spliced.

Possible Issues

MISMATCH The meet of the hitch does not match the meet of the spliced half-hitch

MISMATCH The hoist of the hitch does not match the hoist of the spliced half-hitch

MISMATCH A tether loop was detected while checking the splice

UNKNOWN A tetherline which exceeded the checker's maximum height was detected while checking the splice

Children
rig The rig that is spliced onto the earlier side of the parent rig

lash
0x22586938dc68fa248d582fd7f76adf6f67da179f7baa547e6bc485869545c58d7a

Notes

This is a logical object. Its reference is the identifier of the meet from the bottom half-hitch.

Possible Issues

MISMATCH A tether loop was detected while checking the lash

UNKNOWN A tetherline which exceeded the checker's maximum height was detected while checking the lash

Children
rig The rig that the sibling half-hitch of this lash structure status is lashed onto (the bottom of)

fast-tether
lashing-trie

fast-tether
0x228b730494b3603f48d0650380107043ec621890db2cc421339f713fb38f126ab9

Notes

- This is a logical object. Its reference is the (twist) identifier of the lead in the half-hitch (inside the grandparent rig) which is being lashed (via parent lash) up to a (sibling) rig.

- This object verifies the line connecting the tether of the lead just mentioned, to the fast tether of said same lead

Possible Issues

MISMATCH The tether and all its predecessors back to NULL are loose

MISSING A fast tether could not be found from the provided tether and predecessors

Children

`succession[]`

a list of succession objects in reverse chronological order, starting at the fastener of the half-hitch (from the grantparent rig)

lashing-trie

0x22b4cee32aea047c8aa6236bf95e9ad24991e9d57f5c7d6c9a0b0ef2ff662a21bd

Notes

- This is a logical object. It is referenced by the identifier of the the twist with a missing rigging trie.
- This object ensures that the leadline of the (upper) rig in the lashing has sufficient rigging trie information to properly establish a hoist for the (lower) half-hitch
- The name of this object is subject to change (we don't think it fits our conventions very well, and don't like it).

Possible Issues

MISSING The referenced twist's rigging trie has missing information

A Notation Conventions

A number of data structures are described in this document, as well as functions and relations between them. They are, like code, in the `monospace` font, with `camelCase` names.

Some relations and expectations are defined by reference to iterated functions. Iteration of functions are denoted by superscripts in the usual manner (i.e. `fn0(obj) = obj` and `fnn+1(obj) = fn(fnn(obj))`).

The term *hash* refers to a cryptographic hash function when used as a verb, unless otherwise specified. When used as a noun, it refers to the result of applying a cryptographic hash function, unless otherwise specified.

The term *trie* is used to refer to Merkle tries, as well as other key-value structures that are equally cryptographically secure, such as the pairtrie structure. Additionally, in a trie `trie`, the value associated with a key `key` is denoted as `trie[key]`.

When the value associated with the key in a trie is itself a trie in which a value is being looked up, evaluation is done from left to right – i.e. if `trie[key0]`

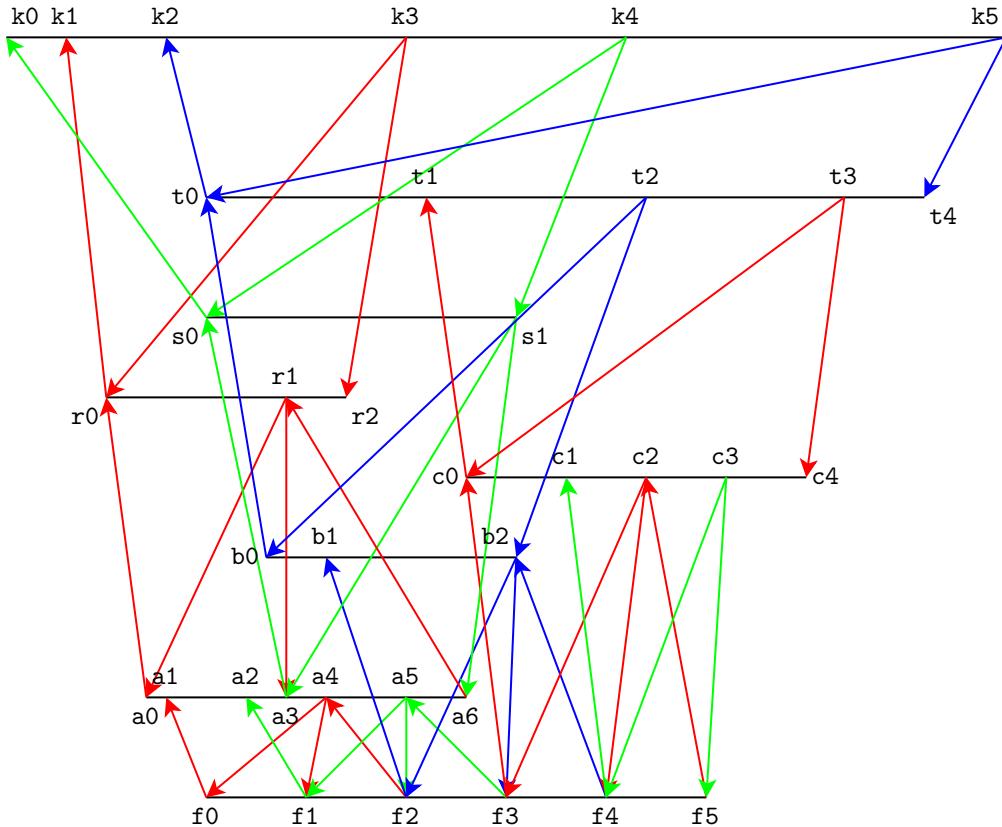
is itself a trie, then the value associated with the `key1` in that trie is denoted `trie[key0][key1]`.

The notation `a|b` is used to for the concatenation of `a` and `b`, as bytestreams, with `a` preceding `b`.

Constant bytestreams are represented in hexadecimal with a preceding `0x`.

B Rig Traversal Example

In order to demonstrate how the rig traversal algorithm verifies the various components of a rig, the following example has been constructed. It includes an assortment of tether transitions and other such elements that are expected to be relatively commonplace in actual implemented rigs (and a few that are expected to be less common, but nonetheless possible). The line containing the twists whose labels begin with `a` will be referred to as the `A` line, etc. The labelled twists are not intended to constitute all the twists in the rig, however any unlabelled twists (except those in the `K` line are required to be loose.



A rig checker shall be provided with all the the atoms needed to construct this rig, and will additionally take as arguments: the (bottom-left) twist `f0` whose

successors are being checked for support, the (bottom-right) twist f_5 whose support as a successor is being checked, the (top-right) twist k_5 supporting the rig, and the (top-left) twist k_0 which is the oldest corkline twist needed to support the leadline.

The steps to be undertaken by a rig checker following the traversal algorithm provided in Section 7.3 are as follows:

1. The (cork)line K is checked (and recorded in the linestore) by following $\text{prev}(s)$ from k_5 to k_0 .
2. The tether of f_0 is followed to a_1 .
3. Because a_1 is not on a line in the linestore, it is on a line that still requires support. Further because it is loose, to get this support we follow $\text{prev}(s)$ until we reach a fast twist a_0 .
4. The tether of a_0 is followed to r_0 .
5. Because r_0 is not on a line in the linestore, it is on a line that requires support. Further, because it is fast, we can get this support by following its tether to k_1 .
6. Because k_1 is in K , we trace forward on K until we reach a twist whose rigging trie meets the criteria to hoist r_0 ; that twist is k_3 .
7. Having k_3 we can now inspect the half-hitch with r_0 as a lead and k_3 as a hoist. This yields R as a footline (which is recorded in the linestore).
8. Because r_0 is on R which is now in the linestore, we trace forward on R until we reach a twist whose rigging trie meets the criteria to hoist a_0 ; that twist is r_1 .
9. Having r_1 we can now inspect the half-hitch with a_0 as a lead and r_1 as a hoist. This yields A' (a subline of A from a_0 to a_3) as a footline (which is recorded in the linestore).
10. Because a_1 is on A' which is now in the linestore, we trace forward on A' until we reach a twist whose rigging trie meets the criteria to hoist f_0 – but A' does not contain such a twist, so we continue along A .
11. Because a_3 is the most recent twist in A' , the tether of a_3 is followed to s_0 .
12. Because s_0 is not on a line in the linestore, it is on a line that still requires support. Further, because it is fast, we can get this support by following its tether to k_0 .
13. Because k_0 is in K , we trace forward on K until we reach a twist whose rigging trie meets the criteria to hoist s_0 ; that twist is k_4 .

14. Having $k4$ we can now inspect the half-hitch with $s0$ as a lead and $k4$ as a hoist. This yields S as a footline (which is recorded in the linestore).
15. Because $s0$ is on S which is now in the linestore, we trace forward on S until we reach a twist whose rigging trie meets the criteria to hoist $a3$; that twist is $s1$.
16. Having $s1$ we can now inspect the half-hitch with $a3$ as a lead and $s1$ as a hoist. This yields A'' (a subline of A from $a3$ to $a6$) as a footline (which is not yet recorded in the linestore).
17. Because the starting twist of A'' is the same as the ending twist of A' (namely $a3$), we verify the splice of the rigs which meet at $a3$ (i.e. those supporting A' and A''). This includes verifying that $a6$ is the post of the hitch with $a0$ as a lead. This verification being done, A' is extended in the linestore by A'' , yielding A in the linestore.
18. Because $a1$ is on A which is now in the linestore, we trace forward on A until we reach a twist whose rigging trie meets the criteria to hoist $f0$; that twist is $a4$.
19. Having $a4$ we can now inspect the half-hitch with $f0$ as a lead and $a4$ as a hoist. This yields F_1 as a footline (which is recorded in the linestore). The meet of this half-hitch $f1$ is not $f5$, so we continue along F .
20. Because $f1$ is the most recent twist in F_1 , the tether of $f1$ is followed to $a2$.
21. Because $a2$ is on A which is in the linestore, we trace forward on A until we reach a twist whose rigging trie meets the criteria to hoist $f1$; that twist is $a5$.
22. Having $a5$ we can now inspect the half-hitch with $f1$ as a lead and $a5$ as a hoist. This yields F'_2 as a footline (which is not yet recorded in the linestore).
23. Because the starting twist of F'_2 is the same as the ending twist of F_1 (namely $f1$), we verify the splice of the rigs which meet at $f1$ (i.e. those supporting F'_2 and F_1). Thus includes verifying that $f2$ is the post of the hitch with $f0$ as a lead. This verification being done, F_1 is extended in the linestore by F'_2 , yielding F_2 in the linestore.
24. The bottom-right of the verified rig is $f2$, not $f5$, so we continue along F and follow the tether from $f2$ to $b1$.
25. Because $b1$ is not on a line in the linestore, it is on a line that still requires support. Further because it is loose, to get this support we follow $\text{prev}(s)$ until we reach a fast twist $b0$.
26. The tether of $b0$ is followed to $t0$.

27. Because t_0 is not on a line in the linestore, it is on a line that requires support. Further, because it is fast, we can get this support by following its tether to k_2 .
28. Because k_2 is in K , we trace forward on K until we reach a twist whose rigging trie meets the criteria to hoist t_0 ; that twist is k_5 .
29. Having k_5 we can now inspect the half-hitch with t_0 as a lead and k_5 as a hoist. This yields T as a footline (which is recorded in the linestore).
30. Because t_1 is on T which is now in the linestore, we trace forward on T until we reach a twist whose rigging trie meets the criteria to hoist b_0 ; that twist is t_2 .
31. Having t_2 we can now inspect the half-hitch with b_0 as a lead and t_2 as a hoist. This yields B as a footline (which is recorded in the linestore).
32. Because b_1 is on B which is now in the linestore, we trace forward on B until we reach a twist whose rigging trie meets the criteria to hoist f_2 ; that twist is b_2 .
33. Having b_2 we can now inspect the half-hitch with f_2 as a lead and b_2 as a hoist. This yields F'_3 as a footline (which is not yet recorded in the linestore).
34. Because the starting twist of F'_3 is the same as the ending twist of F_2 (namely f_2), we verify the splice of the rigs which meet at f_2 (i.e. those supporting F'_3 and F_2). This includes verifying that f_3 is the post of the hitch with f_1 as a lead. This verification being done, F_2 is extended in the linestore by F'_3 , yielding F_3 in the linestore.
35. The bottom-right of the verified rig is f_3 , not f_5 , so we continue along F and follow the tether from f_3 to c_0 .
36. Because c_0 is not on a line in the linestore, it is on a line that requires support. Further, because it is fast, we can get this support by following its tether to t_1 .
37. Because t_1 is on T which is in the linestore, we trace forward on T until we reach a twist whose rigging trie meets the criteria to hoist c_0 ; that twist is t_3 .
38. Having t_3 we can now inspect the half-hitch with c_0 as a lead and t_3 as a hoist. This yields C as a footline (which is recorded in the linestore).
39. Because c_0 is on C which is now in the linestore, we trace forward on C until we reach a twist whose rigging trie meets the criteria to hoist f_3 ; that twist is c_2 .

40. Having c_2 we can now inspect the half-hitch with f_3 as a lead and c_2 as a hoist. This yields F'_4 as a footline (which is not yet recorded in the linestore).
41. Because the starting twist of F'_4 is the same as the ending twist of F_3 (namely f_3), we verify the splice of the rigs which meet at f_3 (i.e. those supporting F'_4 and F_3). Thus includes verifying that f_4 is the post of the hitch with f_2 as a lead. This verification being done, F_3 is extended in the linestore by F'_4 , yielding F_4 in the linestore.
42. The bottom-right of the verified rig is f_4 , not f_5 , so we continue slong F and follow tether from f_4 to c_1 .
43. Because c_1 is on C which is in the linestore, we trace forward on C until we reach a twist whose rigging trie meets the criteria to hoist f_4 ; that twist is c_3 .
44. Having c_3 we can now inspect the half-hitch with f_4 as a lead and c_3 as a hoist. This yields F'_5 as a footline (which is not yet recorded in the linestore).
45. Because the starting twist of F'_5 is the same as the ending twist of F_4 (namely f_4), we verify the splice of the rigs which meet at f_4 (i.e. those supporting F'_5 and F_4). This includes verifying that f_5 is the post of the hitch with f_3 as a lead. This verification being done, F_4 is extended in the linestore by F'_5 , yielding F in the linestore, and completing the verification.

C Glossary

In addition to terms and definitions, this glossary includes a symbol for each term. Translations and terminology shifts may change the particular words we use, but these symbols provide canonical nomenclature for the concepts.

arb An atom containing arbitrary binary data
`0x224629c3aada7d7435c0f80430f92313c265dd9f6522e930d9c64dc77d11809866`

atom An identifier concatenated with its matching packet
`0x228967b176fed39884dc895c6e45bb17c1eb5acf770d9180f6145bff0508c99652`

basic body A simple body shape containing all twist elements except for the satisfactions
`0x2286dcf3cae43636d736de687478bcb9c8fda275628e759b3685e1ab77d3a0fc40`

basic twist A simple twist shape containing a body and a satisfactions trie
`0x226d5fbc31b56883513b96fb17ce9476b0c535040890d00eeb565effaa32d7c5c7`

body	Twist element that is not the satisfactions 0x2295085430dc7c03412d07e4ff91dd5c2c961cccd6ef910f63a408d6f315e41fa3
cargo	Twist body element containing cargo information 0x22da124d85769da9ef497178c90666f9b218bfaf529bed80cfbc3ddc0ed0f37eee
corkline	The line in a rig that provides integrity by supporting the leadline; rhymes with dorkline 0x222db7e19336b76c355ec281232f9aaad367d4a8544f092e4f147ca5bf7344cd8e
fast	A fast twist has a non-NULL value for its tether element 0x2214939b77d4af6d8cbb56599f4b5a0405c02f155da77c5ad801885833623f618a
fastener	The first twist on the topline of a hitch, and tether of the hitch's lead 0x2208632db7be34fc88c72048c3dee66a1de6a9e8b554494bc60a56e94447d58bf
footline	The line in a hitch containing the lead, meet, and post 0x221ff39e9e734a7a758e7a8f36187cf3172ca613d3f904e6d3663c856919624aa1
identifier	A hash algorithm byte, followed by a fixed number of bytes determined by that hash algorithm 0x2295e1873822a7fbe7eda22b4a895593aaf6ae47645741c3884a21b7c83d810061
half-hitch	A hitch without the post, and with a footline terminating at the meet; also a minimal rig 0x222008d2f5c875b46e4f52896ecae0c12f686cc43f57409bde83760fe50388b868
hitch	Fundamental unit of rigging 0x22d0eb6e7af1ca49d58104b3862955078f2e33a1c67645e15459d6808157fd5c88
hoist	The first twist in the topline of a hitch that connects the lead and the meet 0x22863bfd943c67ed29ee7a3e199ac281e93df674809a4a7795317ded106210b30
lash	The result of lashing 0x22586938dc68fa248d582fd7f76adf6f67da179f7baa547e6bc485869545c58d7a
lashing	Vertical composition operator for rigs 0x2234962a88b36217584749c86973c6ad924e58e3b4808287b2e07310ca1b64d47d
lead	The first fast twist on the footline of a hitch; rhymes with mead 0x2288dc9ca034029a13c404ecf210d42698c9971ad78594d72c6e790dc5c4945b2f

headline

The line in a rig that is gaining integrity by support from the corkline;
rhymes with deadline
0x22c0f2930535e522db887925af31c97d204c94eb059869f53c6a548b100ed6daa8

line A sequence of successor twists

0x22366a8aa7e49d39bd21eb95cf26c6b6e899d9213b4c80b638c7d352b68c24954e

meet The second fast twist on the footline of a hitch; rhymes with meat

0x22a94d1bc7016fb4422f0dab55c9c3800b9b98d70cbbb4198a8bc4bff7c15c9bb8

packet

A shape byte and four bytes of length, followed by *length* additional bytes
0x228ecb044099eab83fdb52b6ddc505b82d57376d1afc7cdb8263146ccf2f6a85eb

pairtrie

A simple trie shape containing an ordered list of keys and their values
0x22d9cc0f7b6e15db2cd47642bc32502194ab4737d3eece406e0ebe9320f76677df

post The third and final fast twist on the footline of a hitch; rhymes with most

0x224abe314aa6feb47cdfdc2b07c00c64a2b668f0d9edd4454d95e6c8c01254e73c

predecessor

A twist that precedes another through prev chaining
0x22b02f4d647e52783e5287a421874f60cf05b2f0de7460366e0f2ec6e464a9ee64

prev Twist body element containing the previous twist

0x22158e49e4e6239b1d4649ea36ab7a8cbd0a15ccfb5c26fb5cc313d622894a15d8

requirements

Twist element that be satisfied by a legitimate successor
0x2296b985194b1e7ae9034a021c5fbe2a3ab0eb93bfe3375fec28530037ee9d501a

rig A cryptographic data structure proving the corkline supports the headline

0x22d7357365f862a15d5fc6de991334e96c084bbabfcff1150425c9944fb3937a9e

satisfactions

Twist element that attempts to meet the requirements of its predecessor
0x22bae131f4c554fdf8a8f0bc6bfe255b6592733c6b13b1e9e3748dce5b3fd6f466

shape

A byte at the beginning of a packet describing its data structure
0x22e2b6f5b7b675160d08dea05d32f992c35cf4e219ce9f6f718f14303994b0477e

shield

Twist body element containing the shield information
0x22695cc55218f63f312343fe5a43f8bb8bcde964e70dc2370bb495c2f734d56b68

splice

The result of splicing
0x22fd2e95961c3986ff20ea8371b2406f24fe893a1faaf7759276c9704c1143aa3c

splicing

Horizontal composition operation for rigs

0x228fe955023b969ad90028010fbbd1e43095cf3e7ce76b87a532d6e6c9101d3f3

successor

A twist that succeeds another through inverse prev chaining

0x22292105b6e05dd9eb8d77f276b1af78cd398509089ad43a895d1d7d25c9993673

support

A line L1 supports L2 when L2 has the same integrity as L1

0x2292daacaece9550dd3d7eaf51f7fcfad64b327c6bc64763e7148d9a8dab50dac5

symbol

An atom formed of a single identifier containing 32 arbitrary bytes

0x22274a4d3486b3e29eb56ba22a727d7f39cecbe3d2bca67ca78652d2fa761d17b4

tether

Twist body element containing the tether twist

0x220de6828f486f46b3da3a23ec23900e6d7b709d80373015023e775a5438a404c1

topline

The line in a hitch containing the fastener and the hoist

0x22031c1b1fc93d62b5cac2456f2408a0e4fe6c30560efc82a4270864fbb779e520

trie

A data type for recording key-value pairs

0x22b72533ee7c1fee3ef7a39ad923c16b0fe504d39f5271af1e3639ef6e42ccc4d7

twist

An update to a line, a placemoment in spacetime, a name and a now

0x221267ca19a66c372b90849cb4f807f6c22b1f36c1057f6bba4f5209b12827beae

D Credits and Acknowledgements

Toliver dreamt it. Coward and Toliver worked it, formalized it, specified it. Sulpizi contributed while writing the Clojure implementation, and helped shape status objects. Gravitis contributed while writing the JS implementation. Everson, Fertman, Moir and Levin contributed valuable feedback.

The authors are grateful to TODAQ for their generous financial support of this work.

References

- [1] Kris Coward and D. R. Toliver, *Simple rigs hold fast*, 2022.
- [2] _____, *Simple rigs hold fast*, 2022.
- [3] Adam Gravitis and D. R. Toliver, *Adot objects: Units of meaning in the world of rigging*, Tech. report, 2022.