

telehash

Table of Contents

Introduction.....	3
Glossary	8
Base 32.....	9
Hashnames	10
Packet Format	12
Links	15
Routing.....	19
Mesh Network.....	20
URI Handling.....	21
E3X - End-to-End Encrypted eXchange.....	27
Exchange - Endpoint Encryption State	28
Messages - Asynchronous Content Transport	28
Endpoint Order.....	29
Handshake - Mutual Exchange Creation	30
Channels - Streaming Content Transport.....	32
Reliable Channels	36
Channel Payload Compression	38
Cloaking - Network Obfuscation	40
Cipher Sets	42
Packet Chunking.....	51
Transport Bindings.....	53
UDP Transport	53
TCP Transport.....	54
TLS/SSL Transport.....	55
Channels.....	56
path - Network Path Information	56
peer - Routing Request.....	58
connect - Peer Connection Request.....	59
stream - Reliable Data Streams.....	60
sock - Tunneled TCP/UDP Sockets.....	61
thtp - HTTP Mapping	62

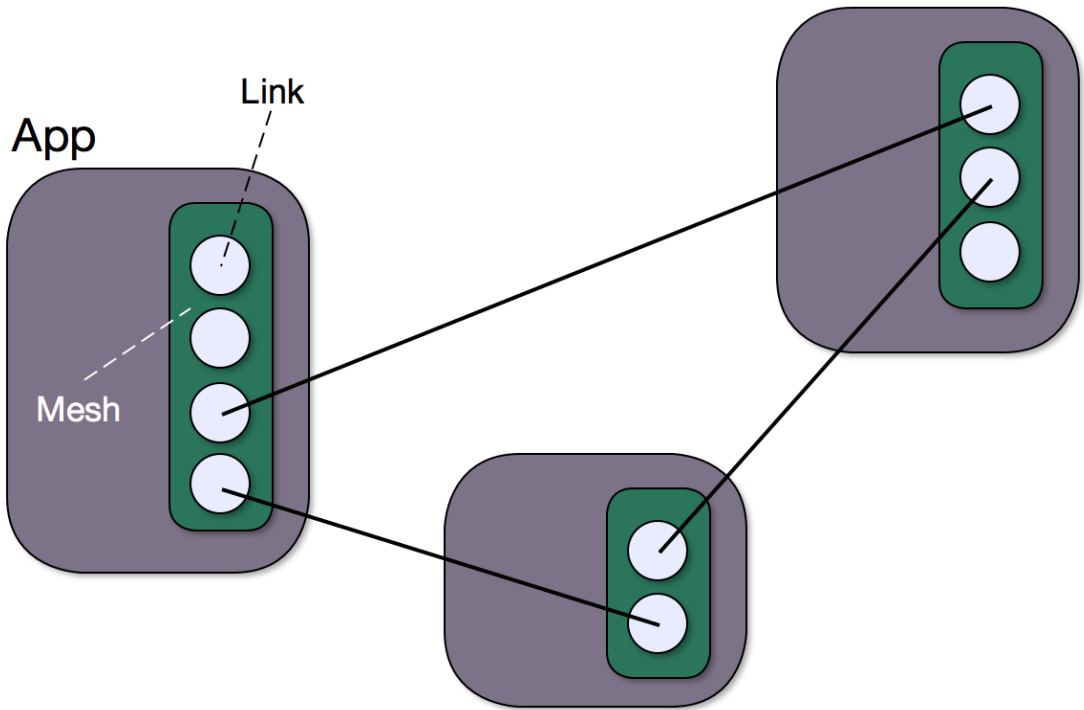
n. *Internet*, from **tele**graph and **has**htable

Introduction

Telehash is a 100% open, secure mesh networking technology with these design principles:

- full end-to-end encryption, all the time
- strict privacy: no content, identity, or metadata is ever revealed to third parties
- maximum app/device compatibility: suitable for embedded, mobile, and web usage
- making privacy the **easy** choice for developers
- flexible transport protocols, for compatibility with existing layers
- native implementations for the widest possible variety of languages/platforms

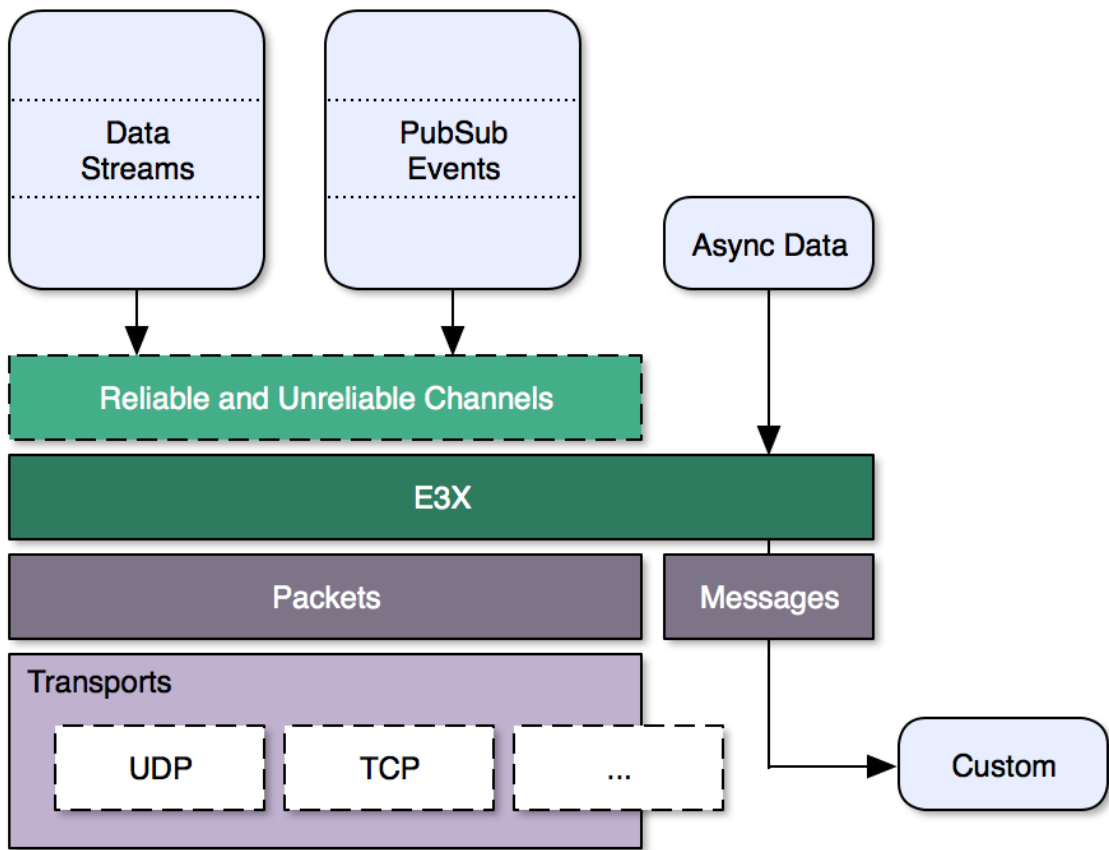
The telehash team includes collaborators from around the world, many of whom were the principal architects of [XMPP](#). It is intended as a next-generation protocol advancing the original goals of Jabber.



The basic architecture contains application or device instances that each maintain a logical mesh which is just a collection of secure links to other instances. Every endpoint has its own view of the mesh, a link between two instances is private only to them and not shared across the mesh.

The current v3 protocol has these high-level properties:

- each endpoint has verifiable unique fingerprint (a hashname), a secure universal MAC address
- provides native tunneling of TCP/UDP sockets, HTTP, object streams, and more
- facilitates asynchronous and synchronous messaging and eventing
- supports an automatic peer discovery mode when available on local transports
- specifies a URI format for initiating new links out-of-band
- supports bridging and routing privately by default and optionally via a public DHT (draft)
- integrates native support for JSON Object Signing and Encryption (JOSE) and OpenID Connect



The protocol stack is separated into two main areas, a lower level end-to-end encrypted exchange (E3X) that handles all of the security primitives and encrypted messages, and higher application-level definitions for managing a mesh of links that support standard channels, transports, and URIs.

Design Philosophy

The primary idea that drove the creation and development of telehash is the belief that any application instance should be able to easily and securely talk to any other application instance or device, whether they are two instances of the same application, or completely different applications. They should be able to do so directly, and in any environment, from servers to mobile devices down to embedded systems and sensors.

By enabling this freedom for developers as a foundation for their applications, telehash enables the same freedom for the people using them - that the user can connect, share, and communicate more easily and with control of their privacy.

The challenges and complexity today in connecting applications via existing technologies such as APIs, OAuth, and REST are only increasing, often forcing fundamentally insecure, centralized, and closed/gated communication platforms. By adopting telehash, an application immediately has a powerful set of open tools for its own needs, but can also easily enable connectivity to and from applications created by others. These tools include the ability to have friends, sharing, feeds, tagging, search, notifications, discovery, transactions, and other social patterns.

Telehash has a fundamental requirement to remain simple and lightweight in order to support applications running on networked devices and sensors. The design goals also include not forcing any specific architectural pattern such as client-server, centralized/federated/distributed, polling/push, REST, streaming, publish-subscribe, or message passing... any can be used, as telehash simply facilitates secure reliable connectivity between any two or more applications in any networking environment.

Usage

The word *telehash* should be pronounced as the root terms would normally be pronounced in the speaker's native dialect.

It is usually written in all lower-case in the sense of a modern version of the word "telephone", both as a generic private communication system that is federated and compatible, and the act of two application instances communicating privately and directly.

Logo



The telehash [logo](#) was designed and contributed by [Jake Ingman](#) and is freely available for apps to use in representing support and current mesh connectivity status.

The official color is [#2a7e60](#).

Change Log

v1

The name and first version of the protocol was established in [2009](#) by [Jeremie Miller](#) and was primarily a DHT-P2P [research project](#).

v2

In 2013 it was significantly updated with a privacy focus as [v2](#).

v3

In 2014 the protocol was refactored and simplified: the wire encryption format was codified as [E3X](#) and the DHT functionality was moved into a separate project called [block-name](#).

- [v3.0.0-stable](#) released on 2015-04-07

Glossary

channel	a virtual stream that allows two <i>endpoints</i> to exchange data reliably or unreliably
chunking	a packet is <i>chunked</i> into smaller pieces for low-MTU or streaming transports
cloaking	method used to hide telehash traffic on the wire by randomizing all data sent
CS	<i>Cipher Set</i> , a collection of crypto algorithms with a given <i>CSID</i>
CSID	<i>Cipher Set ID</i> , predefined hex number identifying a <i>CS</i>
CSK	<i>Cipher Set Key</i> , the public key bytes for a given <i>CSID</i>
endpoint	a participant in the telehash network identified by a single <i>hashname</i>
E3X	<i>End-to-End Encrypted eXchange</i> , a flexible encrypted exchange wire protocol
exchange	the current encrypted session state between two endpoints
handshake	<i>message</i> type used to establish an encrypted <i>session</i> for <i>channels</i>
hashname	an <i>endpoint</i> identifier, calculated from all of its <i>CSKs</i>
LOB	<i>Length-Object-Binary</i> , an encoding format that allows combining JSON and binary data
link	a connection between two <i>endpoints</i> either directly or via a <i>router</i>
mesh	a number of <i>links</i> with active encrypted <i>sessions</i> over any <i>transport</i> ; participants in the mesh are called <i>endpoints</i>
message	an asynchronous encrypted packet between two <i>endpoints</i>
packet	an encapsulation format for JSON and binary data using <i>length object binary (LOB)</i> encoding
router	an <i>endpoint</i> that will facilitate link setup between two other endpoints
transport	underlying network responsible for <i>packet</i> transfer
URI	<i>Uniform Resource Identifier</i> , to enable endpoints to share enough information (<i>hashname</i> , <i>transport</i>) for out-of-band connection setup and references

Base 32

All base 32 encoding used in this document is defined by [RFC 4648](#) with **no padding** and using only the lower case alphabet of `abcdefghijklmnopqrstuvwxyz234567`.

It is used frequently to encode binary 32 byte [SHA-256](#) digests safely for use in JSON and URIs, resulting in a 52 character string, for example:

`kw3akwcy poedvfdqupp ofpujbu7rplhj3vjvmbkvf7z3do7kkq`. It is sometimes also used to encode public key material, small JSON objects, and in implementations to print binary values for debugging.

Base 32 encoding was chosen to maximize compatibility and consistency, such that it is usable in any part of a URI and as DNS labels, and is case insensitive and alphanumeric only. It's only used for small fixed values where interchanging the data safely is more important than efficiency, and is never used to encode dynamic application data over a transport.

Hashnames

A *hashname* is a unique fingerprint to represent the union of one or more public keys of different formats ([Cipher Sets](#)), providing consistent verifiable endpoint addresses when utilizing multiple different public-key algorithms. This enables a compatibility layer for improving the algorithms used so that any application can increase security in newer versions while still being able to represent itself securely to older ones.

A *hashname* can be viewed as a portable, secure, virtual equivalent of a [MAC address](#), since it is a universally unique identifier for a network endpoint that is also self-generated and cryptographically verifiable.

The value is always a [base 32](#) encoded string that is 52 characters long. When decoded it is always a 32 byte binary value, which corresponds to the result of a [SHA-256](#) hash digest. An example *hashname* is

kW3akwcypoeDvfdquuppofpujbu7rplhj3vjvmbkvf7z3do7kkq.

Hashname Generation

A hashname is calculated by combining one or more Cipher Set Keys ([CSK](#)) through multiple rounds of [SHA-256](#) hashing.

The generation has three distinct steps, all of them operating on binary/byte inputs and outputs:

1. Every CSK is identified by a single unique CSID and sorted by it from low to high (e.g. CS1a is lower than CS3a)
2. Each CSK is hashed into *intermediate* digest values
3. Roll-up hashing of the CSIDs and *intermediate* values generates the final 32-byte digest

Any *hashname* generation software does not need to know or understand the Cipher Sets or support the algorithms defined there, it only has to do the consistent hashing of any given set of CSID and CSK pair inputs.

The *intermediate* digest values may be used and exchanged directly instead of the original CSK to minimize the amount of data required to calculate and verify a *hashname* with multiple keys.

Final Rollup

To calculate the *hashname* the *intermediate* digests are sequentially hashed in ascending order by their CSID. Each one contributes two values: the single byte CSID value and the 32 byte *intermediate* digest value. The calculated hash is rolled up, wherein each resulting 32 byte binary output is concatenated with the next binary value as the input. An example calculation would look like (in pseudo-code):

```
1 | hash = sha256(0x1a)
2 | hash = sha256(hash + 0x21b6...f350)
3 | hash = sha256(hash + 0x3a)
4 | hash = sha256(hash + 0x97d8...7101)
5 | final = hash
```

Packet Format

This is a simple encoding scheme to combine any JSON object with any binary data (both are optional) into one byte array, often referred to as a single `packet`. This encoding does not include any total packet size or checksums, and expects the context where it's used to provide those when necessary (see [chunking](#)).

Definition

The wire-format byte array (a packet) is created by combining three distinct parts: the `LENGTH`, an optional `HEAD`, and an optional `BODY`.

The `LENGTH` is always two bytes which are a network-order (big-endian) short unsigned integer that represents the number of bytes for the `HEAD`. When the `HEAD` is greater than 6 bytes the data are always parsed and represented as a UTF-8 JSON object. Any bytes remaining after the `HEAD` are the `BODY` and always handled as binary.

The format is thus:

`<LENGTH> [HEAD] [BODY]`

A simplified example of how to decode a packet, written in Node.js:

```
1 dgram.createSocket("udp4", function(msg){
2   var head_len = msg.readUInt16BE(0);
3   var head = msg.slice(2, head_len + 2);
4   var body_len = msg.length - (head_len + 2);
5   var body = msg.slice(head_len + 2, body_len);
6   if (head_len >= 7) {
7     var json = JSON.parse(head);
8   }
9 });
```

It is only a parsing error when the `LENGTH` is greater than the size of the packet. When successful, parsers must always return five values:

- `HEAD` `LENGTH - 0` to `packet length - 2`
- `HEAD` - undefined/null or binary

- JSON - undefined/null or decoded object
- BODY LENGTH - 0 to packet length - (2 + HEAD LENGTH)
- BODY - undefined/null or binary

LENGTH / HEAD

A LENGTH of 0 means there is no HEAD included and the packet is all binary (only BODY).

A LENGTH of 1-6 means the HEAD is only binary (no JSON).

A LENGTH of 7+ means the HEAD may be a UTF-8 encoded JSON object (not any bare string/bool/number/array value) within the guidelines of [I-JSON](#) beginning with a { and ending with a } character.

If the JSON object parsing fails, the parser must include an error but still return the decoded HEAD/BODY byte structures and lengths.

BODY

The optional BODY is always a raw binary of the remainder bytes between the end of the HEAD and the total length of the packet.

Often packets are attached inside other packets as the BODY, enabling simple packet wrapping/relaying usage patterns (see below).

JSON Web Encryption / Signing (JWE/JWS)

LOB encoding can be used as an optimized serialization of the [JOSE](#) based [JWE](#) and [JWS](#) standards.

The LOB encoding is simply a consistent binary translation of the JWS/JWE compact serialization; there are no semantic changes to the contents in either direction.

JWS

Any [JWS](#) can be mapped to two LOB packets (one attached to another):

- HEAD: {JWS Protected Header (JWT Header)}
- BODY: attached LOB
 - HEAD: {JWS Payload (JWT Claims)}
 - BODY: JWS Signature (binary)

The attached HEAD is treated as a binary octet string when translating. Even though it is frequently JSON it must be preserved as the original bytes for signature validation and non-JSON use cases.

JWE

Any **JWE** can be mapped to three LOB packets (as attached descendents):

- HEAD: {JWE Protected Header}
- BODY: attached LOB
 - HEAD: {JWE JSON (Encrypted Key, IV, Tag, AAD)}
 - BODY: attached LOB
 - HEAD: {optional unprotected header}
 - BODY: JWE Ciphertext (binary)

Example:

```

1  | {"alg":"RSA-OAEP","enc":"A256GCM"}
2  | BODY: {"aad":"","iv":"","tag":"","encrypted_key":""}
3  |   BODY: (no header)
4  |   BODY: ciphertext

```

Links

A *link* is the core connectivity mechanism between two endpoints. An endpoint managing one or more links is referred to as a *mesh*.

Terminology

- **Link CSID** - the highest matching CSID between two endpoints
- **Link Keys** - the one or more CSKs of the other endpoint, at a minimum must include the CSID one
- **Link Paths** - all known or potential path information for connecting a link
- **Link Handshake** - a handshake that contains one CSK and the intermediate hashes for any others to validate the *hashname* and encrypt a response

Link State

Links can be in three states:

- **Unresolved** - at least the hashname is known, but either the Link Keys or Link Paths are incomplete
- **Down** - keys have been validated and at least one path is available (possibly through *router*), but the link is not connected
- **Up** - link has sent and received a handshake and is active

JSON

Many apps use JSON as an easy storage/exchange format for defining and sharing link data. The only required standard for this is that each link is an object with two fields, a *keys* object and a *paths* array. Apps may extend and customize the JSON as needed but should attempt to preserve those two fields to simplify what other tools and libraries can automatically detect and generate.

It's common to have a *hashname* field as well for convenience or as the verified value if only one CSK is stored.

```
1 | {
```

```

2   "keys": {
3     "1a": "aiw4cwmhicwp4lfbsecbdwyr6ymx6xqsli"
4   },
5   "paths": [
6     {
7       "ip": "192.168.0.55",
8       "port": 61407,
9       "type": "udp4"
10    },
11    {
12      "peer": "e5mw...ylka",
13      "type": "peer"
14    }
15  ],
16  "hashname": "frnf...bjdq"
17 }

```

The `keys` object is always a dictionary of at least the single CSID for the link, with all string values being a [base 32](#) encoding of the binary CSK for that given CSID.

The `paths` array is always the list of current or recent [path values](#) and should contain only external paths when shared or a mix of both internal and external when used locally.

JSON Web Key (JWK)

The Link Keys can also be represented in a standard [JWK](#) using a `kty` of `hashname`:

```

1   {
2     "kty": "hashname",
3     "kid": "27yw...lwxa",
4     "use": "link",
5     "cs1a": "an7l...mzlm",
6     "cs3a": "eg3fx...6nia"
7   }

```

The `kid` must always be the matching/correct hashname for the included keys. The `use` value must always be `link` as it can only be used to create links.

The JWK may also contain a `"paths": [...]` array if required. Often the JWK is only used as [authority validation](#) and does not require bundling of the current link connectivity information.

Resolution

Links can be resolved from any string:

1. JSON
2. Direct URI (no fragment)
3. Peer URI (router assisted, with fragment)
4. hashname - peer request to default router(s)

Once resolved, all paths should be preserved for future use. If resolved via a router a peer type path should be preserved referencing that router.

Handshake

In order to establish a link both endpoints must first send and receive an encrypted *handshake*. The minimum required handshake packet is of "type": "link" and the BODY is another encoded packet that contains the sender's hashname details.

The attached packet must include the correct CSK of the sender as the BODY and the JSON contains the intermediate hash values of any other CSIDs used to generate the hashname.

Example:

```
1  {
2    "type": "link",
3    "at": 123456789,
4    "csid": "2a"
5  }
6  BODY:
7    {
8      "3a": "eg3f...6nia",
9      "1a": "ckcz...em6q"
10   }
11  BODY: [2a's CSK binary bytes]
```

The "csid": "2a" is optional when sent as an encrypted message, but required when sent in a peer channel.

Identity (JWT)

The endpoints connected over a link are always uniquely identified by their hashnames. A hostname serves as stable, universally unique, verifiable address, but is not intended to be used as a higher-level identity for an end user or other entity beyond the single instance/device. Once a hostname is generated in a new context, it should be registered and associated with other portable identities by the application.

[OpenID Connect](#) or any service that is able to generate a [JSON Web Token](#) can be used for primary user/entity identification, enabling strongly encrypted communication to be easily coupled with standard identity management tools.

Just as a JWT is sent as a Bearer token over HTTP, it can be automatically included as part of the [handshake process](#) between endpoints with a `"type": "jwt"`. This enables applications to require additional context before deciding to establish a link or apply restrictions on what can be performed over the link once it is established.

Audience

When an [ID Token](#) is generated specifically for one or more known hashnames, the hashname must be included in the `aud` as one of the items in the array value.

Scope

When a client is requesting to establish a new link to an identity, it must include the scope value `link` during authorization.

Claims

An identity may advertise its connectivity by including a `link` member in the [Standard Claims](#). The value must be a valid [URI](#) that can be resolved to establish a link, and any resulting linked hostname must be included in the token's `aud` audience values.

Routing

A `router` in telehash is simply any endpoint that is willing to forward packets between two other endpoints, at least one of which it must already have a [link](#) for. These packets are encrypted such that the routing endpoint cannot decode the contents, but all usage of routing requires a trusted relationship since the routers are at least aware of which two endpoints are connecting to each other.

When a `router` is forwarding the initial handshakes for endpoints first connecting, typically those endpoints will be able to negotiate and establish direct connections immediately and not rely on the router to forward future packets for that session. To support situations in which there is no working direct network path between the two endpoints, the router should continue forwarding packets but may rate-limit them to reduce the impact on its own network.

Any `router` role is an explicit choice by the application, either based on an administrative decision (specific default routers) or a trust relationship (offering to route for friends). These may be through configuration values or automatically discovered by advertising support through a `peer` in the [path](#) channel.

All routing is performed by using a [peer](#) channel to the router, and a [connect](#) channel from the router to the requested endpoint. A `router` never responds to or reveals any known state about the target endpoint to the requesting endpoint.

The use of [URIs](#) may be supported by routers to facilitate easier methods of connecting to new/third-party endpoints.

A [mesh](#) may have one or more default routers which are sent a peer request for every new link connection attempt. Each link may have one or more routers that are signalled from the link itself through a peer path or other application-level relationships, and these routers are used for all future connection attempts for that link.

Mesh Network

A mesh network consists of one or more [links](#), which are active [encrypted sessions](#) between two endpoints over any [transport](#) established by exchanging [handshakes](#).

Each link is identified with a unique [hashname](#). A mesh is private to each endpoint, which has complete control over what links it accepts (there is no automatic sharing of any link state to any other endpoint).

Once handshakes are verified and a link is up, [channels](#) are used to send data over the link between endpoints.

Any link may be set as a [router](#) when it is known that it will provide relaying/bridging to other links. Any endpoint may advertise its [router](#) as a path to other endpoints that may not have the same router. An endpoint may also use a base [URI](#) from a [router](#) as an out-of-band mechanism to establish new links.

Discovery

By default a local endpoint will never respond to any request unless it comes from a link it already knows and trusts. Implementations should support a `discover` mode that can be enabled to temporarily change this behavior and broadcast the endpoint's hashname and keys to any local [network transport](#) that supports discovery.

This mode should be used sparingly so that local networks cannot record what endpoints are available. Typically this is enabled only based on a user behavior ("add a friend" or "pair device", etc) and only for a short period of time. Permanent, local, well-known services/servers that support dynamic association may have it always enabled.

All transports that support discovery should always be listening for incoming `discover` announcements regardless of the discovery state and pass those to the application to evaluate. Discovery does not need to be enabled to receive announcements and see other endpoints, only to announce the local endpoint.

URI Handling

URIs are a convenient method for endpoints to convey connectivity information in any out-of-band medium. This section defines a process for using "link:" URIs (scheme yet to be registered) or other, app-specific URIs to automatically detect associated keys and paths, as well as standard practices for embedding that information in any generated or existing URIs.

Once a URI is successfully resolved and a link is established to an endpoint, the key and path information should be stored and used in place of the URI since a successful resolution only needs to be performed once.

Generation

New URIs that are generated by applications for the sole purpose of establishing a new link take the minimal form of: `link://host:port/?csid=base32`

- `link://` - defaults to `link` but may be any app-specific name for registering custom URL handlers
- `host:port` - the `host` is often an IP address, and the `port` defaults to 42424 if not included
- `cs??=base32` - (optional) a key/value pair for each of the sender's public keys in [base 32](#)

A generated URI will often include additional pathname or query string values as needed by the application.

Examples:

```
chat://127.0.0.1:55772/?cs1a=aof7...supy
```

Router / Peer URI

A common deployment architecture includes a designated stable router that facilitates the connection process with peers. The router can generate an authoritative base URI for peers to re-use and advertise it to them in a [path](#) request. This allows a peer to be reachable via

a URI but still remain private and not share any of its identity information (hashname, keys, or paths).

The router may include its own keys in the query string but may not attach a `#fragment` so that a peer can use the fragment part to include additional data before distributing the URI. The router must always include a value in the base URI to validate the request and internally map it to the peer it was generated for; it should never embed or expose the hashname or other specific details about the peer in the base URI.

The peer endpoint must generate a fragment value that it can use to validate incoming requests and the recipient can use to verify the peer's hashname. This fragment has two parts separated by a `.`, both are base32 encoded byte strings. The first part is always an 8 byte [SipHash](#) digest of all of the second part, using the first 16 bytes of the peer's hashname as the key input to the digest (identical to the [chat channel](#)). The variable bytes remaining in the fragment must be generated by the peer such that they are unique for every URI it shares.

The connecting party might not have a complete fragment, and it only requires the first check digest before the `.` separator. When the URI is resolved the peer must always respond with the full fragment in URI handshake and the connecting party must verify that the hashname is the correct key to generate the digest in the fragment of the complete second part. This ensures that only that peer can correctly link from a specific URI and that the router cannot redirect to another party.

Example URI that uses a router as the base and includes the peer fragment (where X, Y, and Z are base32 values):

```
link://127.0.0.1/?sid=42&cs1a=X#Y.Z
```

When a URI is processed that contains a fragment it generates a new [peer](#) request to the router that includes the `"uri": "..."` (without the fragment) for the router to validate and resolve to the right peer. The peer request must still also include a full URI handshake to be forwarded directly to the peer for its own validation.

Processing:

When processing a new URI, the steps are:

1. detect included keys in the query string and derive hashname

- [optional] fallback to discover keys via WebFinger
 - fallback to resolve the canonical hostname to discover keys via DNS SRV
 - fallback to discover keys via OpenID Connect Discovery
2. generate paths for all supported transports with any resolved IP and port
 3. create a link with the keys and path(s) including a URI handshake
 4. if there's a fragment hashname, issue a `peer` request over the link to it including a URI handshake
 - process any response URI handshake with a validating fragment as the final resolution

Embedded Keys

The CSKs for an endpoint may be included in the query string of any existing URI.

Each CSK is included as an individual key/value pair where the key is the CSID in the format `cs??` (`cs1a`, `cs2a`, etc) and the value is always the base32 encoded key bytes.

Embedded Paths

The current paths may also be included in the query string of any existing URI. Each available path has its JSON object base32 encoded as the value and is included with a common `paths` key, multiple paths have the same key.

When the paths match the ones generated from the hostname in the URI it is not necessary to include them in the query string.

Example paths:

```

1  [
2      {
3          "url": "http://192.168.0.36:42424",
4          "type": "http"
5      },
6      {
7          "ip": "192.168.0.36",
8          "port": 42424,
9          "type": "udp4"
10     },

```

```

11     {
12         "ip": "fe80::bae8:56ff:fe43:3de4",
13         "port": 42424,
14         "type": "tcp6"
15     }
16 ]
17 URL: proto://host/path?
18     key=value
19     &paths=pmrh...yce7i
20     &paths=pmrg...qrh2
21     &paths=pmrg...rcpu

```

Generated Paths

The canonical hostname (or IP and port) of a URI or the resolved SRV `port` and `IP` should be treated as a potential path for all available transports for an endpoint. Handshakes should be sent to the given address in every transport supported that can use an IP and port, including UDP, TCP, TLS, and HTTP(S).

Link Discovery

When an endpoint's keys cannot be included directly in the URI they may be discovered via automated techniques from other parts of the URI.

DNS Links

SRV records always resolve to a hashname-prefixed host, with TXT records returning all of the keys.

- `_link._udp.example.com. 86400 IN SRV 0 5 42424 uvab...hv7g.example.com.`
- `uvab...hv7g IN A 1.2.3.4`
- `uvab...hv7g IN TXT "1a=base32"`
- `uvab...hv7g IN TXT "2a=base32"`
- `uvab...hv7g IN TXT "2a2=base32"`
- `uvab...hv7g IN TXT "3a=base32"`

If a key's base32 encoding is larger than 250 characters (TXT limit), it is broken into multiple TXT records with the `CSID` being numerically increased so that it can be consistently reassembled.

No other DNS record type is supported, only SRV records resulting in one or more A and TXT records.

WebFinger Links

Use [WebFinger](#) against the canonical hostname, passing the given URI in as the resource and a `rel` value of `http://telehash.org/link`. If successful, it will result in a valid `href` that must return the standard [JSON link](#) description format.

```
GET https://example.com/.well-known/webfinger?
  resource=http://example.com/~user1
  &ref=http://telehash.org/link
{
  "subject": "http://example.com/~user1",
  "links" : [
    {
      "rel" : "http://telehash.org/link",
      "href" : "https://www.example.com/~user1/link.json"
    }
  ]
}
```

```
GET https://example.com/~user1/link.json
{
  "keys":{...},
  "paths":[...]
}
```

OpenID Connect Discovery

One or more hashnames may be advertised as part of [OpenID Connect Discovery](#) by simply including their [JWK](#) in the `jwks_uri` response. If there are multiple keys in the response only the first one in the array should be used to resolve the URI.

The discovery endpoint may also be used as a default authority to validate against for incoming unknown hashnames before responding to them.

URI Handshake

When a URI is the source of a new link, a `"type": "uri"` [handshake](#) should be sent including the original URI.

Example:

```
1 {  
2   "type": "uri",  
3   "uri": "https://example.com/link?ref=42#u8kb..."  
4 }
```

E3X - End-to-End Encrypted eXchange

E3X is a flexible end-to-end encrypted wire protocol, a specification for applications to route interoperable packetized content over any transport while protecting the privacy of those communications from network monitoring.

It is designed to be used as a low-level software library that can be embedded in any app. It exposes *all* trust decisions to the app layer; zero information or metadata is revealed to any network or endpoint without explicit instructions from the app.

All of the cryptographic primitives used in E3X are defined as a [Cipher Set](#), allowing for applications to select for different resource or security requirements as needed.

E3X defines asynchronous [messages](#) and synchronous [channels](#) managed using an [exchange](#). Each [exchange](#) has mutual session state established through explicit [handshakes](#).

An [exchange](#) is created by combining keys for the local endpoint (one or more, depending on what Cipher Sets it supports) and has another endpoint's public key(s). Once created, the [exchange](#) can then be used to create encrypted [messages](#) and generate [handshakes](#) in both directions.

Once the [handshakes](#) have been received and verified, encrypted [channels](#) can stream reliable or unreliable data between the two connected endpoints. All data is encoded as [packets](#) both before (application layer) and after encryption (wire transport layer).

Comparisons

These are similar low-level encrypted wire protocols:

- [OTR](#), ([spec](#))
- [CurveCP](#)
- [QUIC](#) ([spec](#))
- [SRTP](#) ([spec](#))
- [MinimalT](#)

No existing protocols met the requirements for telehash to minimize all metadata revealed to any network while also supporting both the sync and async communication patterns.

Exchange - Endpoint Encryption State

An `exchange` is the name of the current encryption state to one remote endpoint, a pairing between the local endpoint's private key and the remote endpoint's public key.

When a new `exchange` is created it will generate a unique public `ROUTING TOKEN` value for that current instance, any incoming encrypted channel packets with that matching value can then be delivered to the appropriate `exchange`.

The `exchange` is used to generate and process all `messages`, `handshakes`, and `channels`. It can only generate channel packets after it has processed at least one handshake from the remote endpoint to set up the required synchronous encryption state.

An `exchange` internally keeps track of the current/best `at` value for the handshakes, validates incoming channel IDs, and generates new outgoing channel IDs.

Messages - Asynchronous Content Transport

Message packets are for encrypting small amounts of content to other entities without requiring a synchronized `exchange`, such that the recipient can process them at any point in the future. They are used primarily for creating ephemeral `handshakes` to establish synchronous `channel` encryption with forward secrecy guarantees.

Content sent in a message has a lower level of privacy since there is no forward secrecy protection like channels provide, if the recipient is compromised any past messages sent to it can be decrypted. Messages should only be used minimally for temporary or non-secret data, all private or sensitive content should be sent over channels. Messages should also never be stored at rest, logged, or archived.

Messages define how to encrypt the packets but have no required internal structure (unlike channels). There is a larger overhead for encrypted message packets as they must always include the ephemeral public key information used and often require additional computation as well.

An [exchange](#) may be created on demand just to generate/process one or more messages and not used for channels, but since messages are asynchronous they do not require the exchanges to be in sync to operate.

The size of an encrypted message is determined by the application and context in which it is used, handshakes are usually small (<1400 bytes to maximize transport compatibility) and any messages intended to be sent over a transport with a low MTU may need to use [chunked encoding](#) as the BODY of multiple messages.

Packet Encryption

All message packets are encrypted using a cipher as determined by the [Cipher Set](#) in use for the exchange. The encrypted (OUTER) packets must have a HEAD of length 1 to identify the CSID and the encrypted contents as the binary BODY.

Once decrypted they result in an INNER packet with a structure that is determined entirely by the application. It is common practice for applications to use a "type": "value" on the INNER JSON similarly to channel packets, but not required. All INNER packets should contain a mechanism for the recipient to determine recency and thus ensure that the ephemeral keys already used can be invalidated and not-reused if required for forward secrecy.

Stable Prefix

All message packets generated from one exchange must have at least the first 16 bytes remain fixed for the lifetime of that exchange to be used for generating the [ROUTING TOKEN](#) and validation caching.

Endpoint Order

In order to prevent conflicts between any two endpoints, uniqueness is guaranteed by comparing the public keys of their agreed upon [cipher set](#).

The binary value of each endpoint's public key is compared and they are placed in high-to-low sorted order. To achieve this the two public keys can be visited bit by bit, the first key that at a given position has a 1 (*higher*) where the other has a 0 (*lower*) is the one that has *higher* order.

Based on this comparison the two are labeled as ODD (*high*) endpoint and an EVEN (*low*) endpoint.

This mechanism ensures that that both endpoints will mutually agree on their label consistently and without communication.

Handshake - Mutual Exchange Creation

A handshake is one or more encrypted [messages](#) sent between two endpoints in order to validate their identity and establish mutual [exchange state](#) to use for [channels](#). At a minimum at least one handshake message must be both sent and received in order for the session to be created, with both endpoints verifying that it is always the most current one.

Applications may send or expect more than one handshake message for additional authentication and authorization requirements beyond the basic endpoint key exchange. New handshakes are also triggered automatically for existing sessions as needed by the transport(s) in use to verify that the network paths are still valid and/or maintain any NAT mappings.

The resulting size of encrypted handshake packets vary by which [Cipher Sets](#) are used combined with the type of inner packet, typically it ranges from ~70 to ~1100 bytes.

Resend/Timeout

After a new handshake is generated and delivered it should be resent verbatim at 1 second, 3 seconds, 7 seconds, and again at 15 seconds unless there is a valid handshake response. After 30 seconds with no response the exchange should be timed out, considered invalid and all related state removed.

At any point the transport being used to deliver packets may generate a keepalive handshake request which will start this process.

Message Types

Any decrypted handshake message is identified with a "type": "... " string value, that if not included in the header must be defaulted to the type of "link". Only one unique type may exist concurrently (same at value) with any handshake process.

Known types include:

- **link** - to establish or keepalive a link
- **jwt** - message BODY is a JSON Web Token encoded packet
- **uri** - a URI was used to generate this handshake and it is included as the "uri": "... " value.
- **tx** - message BODY is a raw bitcoin transaction

Sequencing with at

All decrypted handshake messages must contain an "at":123456 with a 64 bit positive unsigned integer value to determine the newest generated handshake from either endpoint. There is no requirement for the at value to be the current time, in sync, or accurate, only that it increases on all subsequent handshakes in the future from the last highest known value.

Multiple messages as part of one handshake must all have the same at value and different types, only one message per type with the highest at is used.

The at value determines if an incoming handshake is the most current and if the recipient needs to respond. The last bit in the at must always match the **order** value of the sender, if they are ODD it must be a 1 (and 0 if they are EVEN) to guarantee that no two endpoints can choose the same at independently.

When an at is received that is higher than one sent, new handshake message (or messages) must be returned with that matching highest at value in order to inform the sender that their handshake is confirmed. Upon receiving and confirming a new at, any pending channel packets that may have been waiting to send may be flushed/delivered.

When first creating a handshake, the sender should make every effort to always choose a higher at than any they may have sent in the past. Most can just use local **32-bit epoch** as this value, but when not available (e.g., on embedded systems) they should locally store the last sent at and always increase it.

If the maximum at value is ever reached/used the two hashnames cannot send any more subsequent handshakes and will no longer be able to communicate. In this case, either side must generate a new hashname to start over.

Routing Token

The handshake determines the exchange's `ROUTING_TOKEN` value, generated by SHA-256 hashing the first 16 bytes of the encrypted outer message `BODY`, then using the first 16 bytes of the digest output. The `ROUTING_TOKEN` is not used cryptographically and is only a unique value to assist the two endpoints and any routing parties on mapping to a known exchange session.

Each [Cipher Set](#) must structure their messages such that the first 16 bytes of the `BODY` are unique and remain stable for the lifetime of the exchange, typically being the exchange's ephemeral public key bytes.

Any handshake with a different `ROUTING_TOKEN` and a higher `at` of an existing exchange for the same endpoint must clear any cached handshake messages or state stored for that exchange.

Handling Multiple Messages

Immediately upon receiving a valid higher or equal `at` for any handshake message type, that message should be cached and replace any previous matching type with a *lower* `at` only (messages with matching `at` and `type` values must be discarded). The message should also then be grouped with any other received messages with the same `at` and delivered to the application to process and validate.

If an application requires multiple message types it simply waits until the sufficient types arrive and then processes/validates them. Application-invalidated handshakes must never be responded to so that the endpoint does not advertise its existence except to explicitly trusted/validated endpoints.

At any point the application may provide updated handshake message types to be sent in a new handshake process. When the transport requests an updated handshake, the last known/provided message types are updated with a new `at` and re-sent.

Channels - Streaming Content Transport

All streaming data sent between two endpoints in an exchange must be part of a `channel` packet. Every channel has an integer id included as the `c` parameter in the

JSON and managed by the current [exchange](#). See [Channel IDs](#) for details on how they are selected/handled.

A channel may consist of only one outgoing initial packet and one response to it, or it may be long-lived with many packets exchanged using the same "c" identifier (depending on the type of channel). Channels are by default unreliable with no retransmit or ordering guarantees, and may optionally be created as [reliable](#) to provide ordering and retransmission functionality.

An `"end": true` always signals the last *content* packet being sent for any channel type; for reliable channels there may be subsequent acks/retransmits after the end.

Packet Size Default

Channel packets should always be a maximum of 1400 bytes or less each, which allows enough space for added variable encryption, token, and transport overhead to fit within 1500 bytes total (one Ethernet frame). Larger data blocks should use reliable channels for sequencing and reassembly, and transports with a fixed MTU lower than 1400 should use [chunked encoding](#) by default.

A channel library should provide a `quota` method per packet for the app to determine how many bytes are available within the 1400 limit, and app-specific channel logic can use this to break larger data into packets. In special cases (such as with a local high bandwidth transport) when the transport MTU is known, the app or custom channel logic may ignore this and send larger/smaller packets.

Packet Encryption

All channel packets are encrypted using a stream cipher as determined by the [Cipher Set](#) in use for the [exchange](#). The encrypted (OUTER) packets must have a `HEAD` of length 0 and the encrypted contents as the binary `BODY`.

Once decrypted they result in an `INNER` packet that must always contain valid JSON (have a `HEAD` of 7 or greater).

Decrypted Packets

Base parameters on channel packets:

- `"type": "value"` - A channel always opens with a `type` in the first outgoing packet to distinguish to the recipient what the name/category of the channel it is. This value must only be set on the first packet (called the *open packet*), not on any subsequent ones or any responses.
- `"end": true` - Upon sending any content packet with an `end` of `true`, the sender must not send any more content packets (reliability acks/resends may still be happening though). An `end` may be sent by either side and is required to be sent by *both* to cleanly close a channel, otherwise the channel will eventually close with a timeout.
- `"err": "message"` - As soon as any packet on a channel is received with an `err` it is immediately closed and no more packets can be sent or received at all, any/all buffered content in either direction must be dropped. Any `err` packets must contain no channel content other than additional error details. Any internal channel inactivity timeout is the same as receiving an `"err": "timeout"`.
- `"seq": 1` - A positive integer sequence number that is only used for and defined by [reliable](#) channels. It must be sent in the first open packet along with the `type`. It is an error to send/receive this without using reliability on both sides.

An example unreliable channel start packet JSON for a built-in channel:

```

1  {
2      "c":1,
3      "type":"path",
4      "paths":[...]
5  }
```

An example initial reliable channel open request:

```

1  {
2      "c":2,
3      "seq":1,
4      "type":"hello",
5      "hello":{"custom":"values"}
6  }
```

Channel States

A channel may only be in one of the following states:

- `OPENING` - the initial channel open packet containing the `type` has been sent or received, but not confirmed or responded to yet and will time out in this state
- `OPEN` - the channel open packets have been both sent and received and it will not timeout unless the exchange itself does or unless reliability fails
- `ENDED` - a packet containing an `"end": true` has been received and no further content will be delivered for this channel, it will be timed out

These are the states that E3X manages, if an application requires additional states (such as when one party ended but the other hasn't) it must track them itself. Any channel having received or sent an `err` is immediately removed after processing that packet and no more state is tracked (i.e., there is no channel error state).

Any channel that is in the `ENDED` state and has also sent an `end` is no longer available for any sending/receiving, but internal state will be tracked until the channel timeout for any necessary reliability retransmits/acknowledgements.

Channel IDs

A Channel ID is a *positive* integer (`uint32_t`) from 1 to 4,294,967,295 and is determined by the sender and then used by both sides to send/receive packets on that channel. In order to prevent two endpoints from picking the same `c` value they choose them based on their [order](#): the `ODD` endpoint uses odd numbers starting with 1, and the `EVEN` endpoint uses even numbers starting with 2. 0 is never a valid ID.

When a new channel is created, the ID must be higher than the last one the initiator used, they must always increment. Upon receiving a new channel request, the recipient must validate that it is higher than the last active channel (note: switches must still allow for two new channel requests to arrive out of order).

When a new exchange is established, it errors any `OPEN` channels and sets the minimum required incoming channel IDs back to 1.

If the maximum ID is reached the exchange must be regenerated, resetting it back to 1.

Timeouts

Every channel is responsible for its own timeout and may have a different value than others. A timeout occurs whenever the channel is in `OPENING` or `ENDED` state or when any packet has not been ack'd for reliable channels.

Any channel that is in `OPEN` state will not trigger a timeout individually since the exchange as a whole will timeout if the connection is lost based on the network transports in use. Those timeouts independently occur at a higher level for the overall exchange when the handshake process fails and do not use any channel timeout values.

Reliable Channels

Channel packets are by default only as reliable as the underlying transport itself is, which often means they may be dropped or arrive out of order. Most of the time applications want to transfer content in a durable way, so reliable channels replicate TCP features such as ordering, retransmission, and buffering/backpressure mechanisms. The primary method of any application interfacing with an E3X library is going to be through starting or receiving reliable channels.

Reliability is requested on a channel with the very first packet (that contains the `type`) by including a `"seq": 1` with it, and a recipient must respond with an `err` if it cannot handle reliable channels. Reliability must not be requested for channel types that are expected to be unreliable.

seq - Sequenced Data

The requirement for a reliable channel is always including a simple incrementing `"seq": 1` positive integer value on every packet that contains any content (including the end). All `seq` values start at 1 with the open and increment per packet sent when it contains any data to be processed, with a maximum value of 4,294,967,295 (a 32-bit unsigned integer).

A buffer of these packets must be kept keyed by the `seq` value until the recipient has responded confirming them in an `ack`. When the buffer is nearing full or new packets are being dropped, a `miss` should be sent to indicate what is missing and the capacity left.

The receiving app logic must only process sequenced packets and their contents in order, any packets received with a sequence value that is older than already processed ones must be dropped, and any out of order must either be buffered or dropped depending on local resources available.

ack - Acknowledgements

The "ack":1 integer is included on outgoing packets as the highest known seq value confirmed as *delivered to the app* (as much as is possible to confirm immediately). What this means is that any library must provide a way to send data/packets to the app using it in a serialized way, and be told when the app is done processing one packet so that it can both confirm that seq as well as give the app the next one in order. Any outgoing ack must be the last processed seq so that the sender can confirm that the data was completely received/handled by the recipient.

If a received packet contains a seq but does not contain an ack then the recipient is not required to send one for the given seq while it's still processing packets for up to one second. This allows senders to manage their outgoing buffer of packets and the rate of acks being returned, and ensures that an ack will still be sent at a regular rate based on what is actually received.

An ack may also be sent in its own packet ad-hoc at any point without any content data, and these ad-hoc acks must not include a seq value as they are not part of the content stream and are out-of-band.

When receiving an ack the recipient may then discard any buffered packets up to and including that matching seq id, and also confirm to the app that the included content data was received and processed by the other side.

miss - Missing Sequences

The "miss":[1,2,4] is an array of positive delta integers and must be sent along with any ack if in the process of receiving packets there are missing sequences. The array entries each represent a seq value calculated as the delta from the previous entry, using the accompanying ack as the initial base to start calculating from.

The last entry in the array always represents the seq id that the recipient will start dropping packets at; it is the maximum capacity of the incoming unprocessed packet buffer. Whenever the buffer is over 50% full the recipient should send a miss to indicate the capacity left even if there are no other missing packets. When the sender gets a miss it should always cache the total delta number as the maximum window size and never send packets with a higher seq than the last received ack+delta.

Upon receiving a `miss` the recipient should resend those specific matching calculated `seq id` packets in its buffer. If the missing `seq` is signaled in multiple incoming packets quickly (which happens often), the matching packet should only be resent once until at least one second has passed.

The `miss` recipient can make no assumptions about the sender's state of any `seq ids` higher than the `ack` and not included in the array. It can only use the values included as a signal that those sequences are missing.

miss delta encoding example

Given the raw list of missing `seq ids` `[78236, 78235, 78245, 78238]` and `"ack": 78231`.

1. Sort the original list of missing `seq ids`:
`[78235, 78236, 78238, 78245]`
2. Calculate the difference between all subsequent ids (including the `ack`).
`[(78235 - 78231), (78236 - 78235), (78238 - 78236), (78245 - 78238)]`
`[4, 1, 2, 7]`
3. If the incoming max buffer size is 20 packets, append the highest acceptable `seq` (78251) as a final delta (`78251 - 78245`).
`[4, 1, 2, 7, 6]`
4. Deliver the final delta encoded `miss` array.

Channel Payload Compression

Since channel packets are the most frequent and have a set of fixed well-known key/values in their JSON headers, both endpoints may support optional channel compression encoding to minimize the resources required.

This is important in embedded/device networks where the MTU is small (e.g., Bluetooth LE and 6lowpan), and may improve performance in other edge cases with frequent small packets.

z Link Handshake Signalling

To indicate support of a channel payload compression any endpoint may include a **z** key with an unsigned integer value in the link handshake. The value 0 is the default and signals no support.

The **z** value indicates how to decode/interpret the channel payload bytes immediately after decryption. After any alternative processing the resulting value must still always be identical to a LOB packet with a JSON header and binary BODY, it is only to minimize encoding and not for use to include additional data in a payload.

Both endpoints must include identical **z** in a confirmed link handshake in order for it to be enabled on any channel packets using the resulting keys, and only that type of channel payload is supported when enabled. Because there is no negotiation or signalling of support for multiple values, future **z** values will be defined that combine multiple techniques when necessary.

0 LOB encoded (default)

All channel payload bytes are [LOB encoded](#).

1 CBOR encoded

The value 1 signals support of [CBOR](#) payloads, the bytes are interpreted as a stream of CBOR values instead of LOB encoding.

- first value is always channel id ("c", unsigned int)
- [optional] byte string of a payload LOB packet
- [optional] map of additional key/value pairs
- [optional] text value is the "type" string value
- [optional] unsigned int is a "seq" value
- [optional] array is the "ack" and "miss" unsigned int values, ack is always the first value in the CBOR array

Decoding

When processing CBOR the result is always a regular LOB packet with a JSON header.

1. decode the channel id

2. if a byte string follows it is processed as the source LOB, if not then generate a blank/empty LOB packet
3. set the "c":id in the packet JSON to the channel id from 1.
4. if a map follows, its key/value pairs must be processed and only text keys and text or number values are used, each one being set in the packet JSON
5. if a text value follows, it is set as the "type":value in the JSON
6. if an unsigned int follows, it is set as the "seq":value in the JSON
7. if an array follows, it must be processed and only unsigned int values are used, the first one is always set as the "ack":value and all other entries in the array are the "miss":[1,2,3] in the JSON.

Examples

JSON {"c":1,"type":"open"} (21) [CBOR](#) (6):

```
01          # unsigned(1)  // c
64          # text(4)      // type
6f70656e   # "open"
```

JSON {"c":2,"seq":22,"ack":20,"miss":[1,2,20]} (41) [CBOR](#) (7):

```
02          # unsigned(2)  // c
16          # unsigned(22) // seq
84          # array(4)
14          # unsigned(20) // ack
01          # unsigned(1)
02          # unsigned(2)
14          # unsigned(20)
```

2 DEFLATE encoded

All channel payload bytes are encoded/decoded with [DEFLATE](#) before/after encryption. The uncompressed bytes are always a normal LOB packet.

Cloaking - Network Obfuscation

In situations where the network may be performing any packet filtering or inspection it is important to add as much random noise as possible to all bytes sent across an untrusted/

unencrypted transport. Cloaking is a simple and efficient technique that can be used on any transport and is the default for all unencrypted ones by default (such as TCP and UDP).

The cloaking technique simply requires an extra processing step that adds a random number of 8 byte nonces to every packet and randomizes 100% of the bytes on the wire. It makes large-scale pattern identification techniques significantly more difficult, but is not a guarantee that individual packets cannot be targetted. Future designs will continually increase this difficulty.

Per-Packet

Due to all encrypted packets beginning with single zero byte (0x00) when sent on the wire (since they have no JSON encoded), cloaking uses a first byte that is any non-zero value (0x01 to 0xff).

Cloaking is performed using the [ChaCha20 cipher](#) and choosing a random nonce of 8 bytes that does not begin with 0x00. The key is a fixed well-known 32 byte value of d7f0e55546241b2a944ecd6d0de66856ac50b0baba76a6f5a4782956ca9459a (shown as hex encoded), which is the SHA-256 of the string `telehash`.

The resulting cloaked packet is the concatenation of the 8-byte nonce and the ChaCha20 ciphertext output. Once decloaked, the ciphertext should be processed as another packet, which may be a raw encrypted packet (0x00) or may be another cloaked one. A random number of multiple cloakings should always be used to obfuscate the original packet's size.

Noise Floor

Any endpoint may wish to obfuscate not only the individual packets but also the rate or timing of any transmission. A noise floor can be established by generating random but valid cloaked packets of a minimum size and sending them continuously at a fixed rate to the recipient. When real packets are waiting to be sent they are swapped out and sent instead of the next larger noise floor packet and cloaked multiple times to match the noise floor size.

Accept both

All implementations must support receiving both cloaked and uncloaked packets, and the default for any un-encrypted transport should always be cloaking enabled. The initial sender determines when to send un-cloaked packets on any transport, but when receiving a cloaked packet any sender should always respond with cloaked packets as that may be the only way to ensure they are transmitted.

Cipher Sets

A Cipher Set (CS) is a group of crypto algorithms that are used to implement the core security functions as required by E3X. Multiple sets exist to allow an evolution of supporting newer techniques as well as adapting to different system and deployment requirements.

A set always contains an endpoint public key cipher, an ephemeral public key cipher (for forward secrecy), and an authenticated streaming cipher. Often a set uses the same public key algorithm for both the endpoint and ephemeral ciphers with different keys for each.

Cipher Set Key (CSK)

Each set can generate a single public key byte array called the Cipher Set Key (CSK) that is shared to other entities in order to generate outgoing or validate incoming messages.

The CSK is a consistent opaque value intended for use only by a given CS. It must be treated as an arbitrary *binary octet string* when transferred, imported, or exported.

Cipher Set ID (CSID)

Each CSK is identified with a unique identifier (CSID) that represents its overall selection priority. The CSID is a single byte, typically represented in lower case hex. The CSIDs are always sorted from lowest to highest preference.

Two endpoints must always create [exchanges](#) to each other using the highest common CSID between them. Apps may choose which one or more CSIDs they want to support when they create an endpoint and know that a lower one will only ever be used to communicate with other endpoints that only support that CS.

Every CS requires a strong/secure random number generator in order to minimally function, some of them may have additional entropy requirements during CSK generation.

The 0x00 CSID is not allowed and always considered invalid.

Any CSID of 0x0* (0x01 through 0x0f) is for experimental use when developing custom Cipher Sets and should not be used in production.

Reserved

All CSIDs with the mask of 11111000 (0x18 through 0x1f, 0x28 through 0x2f, etc) are reserved and their usage is specified in this table:

CSID Status	Crypto	Uses
CS1a	Active ECC-160, AES-128	Embedded, Browser
CS1b	Draft ECC-256, AES-128	Hardware-Accelerated
CS1c	Draft ECC-256k, AES-256	Bitcoin-based Apps
CS2a	Active RSA-2048, ECC-256, AES-256	Server, Apps
CS2b	Draft RSA-4096, ECC-521, AES-256	High-Security
CS3a	Active Curve25519, XSalsa20, Poly1305	Server, Apps

Custom

Any CSID with the mask of 11110111 (0x10 through 0x17, 0x20 through 0x27, etc) are for custom application usage, these Cipher Sets definitions are entirely app-specific. Implementations are responsible for ensuring that the custom CSIDs match their security preferences.

See the [JOSE-based](#) mapping draft for example custom CSIDs.

Cipher Set 1a

This is a minimum lightweight profile to support embedded devices and low resource environments. It has additional constraints in place to minimize the code size and bytes on the wire (prioritized over speed) while maintaining a modern level of privacy.

The base algorithms used in this set are chosen to be readily implementable on embedded hardware (8bit 16mhz 32k AVR in < 1 second) and are considered minimum-grade security:

- **ECC secp160r1** - small key sizes, balance of relatively strong crypto and still supportable with low CPU
- **HMAC-SHA256** - common implementations available for embedded environments
- **AES-128-CTR** - low impact streaming cipher, many implementations including hardware ones

Keys

When generating an endpoint including CS1a, the public/private keypair is an ECC secp160r1 curve and the binary public key format is [compressed](#), 21 bytes in length.

One key is generated permanently to identify the local endpoint, and one ephemeral key is generated on demand for every exchange created.

Message BODY

The BODY of any message packet is binary and defined with the following byte sections in sequential order:

- KEY - 21 bytes, the sender's ephemeral exchange public key in compressed format
- IV - 4 bytes, a random but unique value determined by the sender
- INNER - the AES-128-CTR encrypted inner packet ciphertext
- HMAC - 4 bytes, the calculated HMAC of all of the previous KEY+IV+INNER bytes

By performing ECDH with the received ephemeral key and the recipient's identity key, the resulting 20 byte secret is SHA-256 hashed and folded once to create the 16 byte AES-128 key along with the given IV (right-zero-padded to 16 bytes).

Another ECDH is performed with the sender and recipients identity key, and that 20 byte secret is combined with the given IV and input to HMAC-SHA256 of the entire BODY bytes (KEY+IV+INNER) minus the HMAC, then folded three times to get the 4 byte verification value that must match/be the last 4 bytes in the BODY.

Channel Setup

An exchange can generate or process channel packets once it has received a valid handshake. The channel encryption is generated by using ECDH with the sent and received ephemeral keys from the handshakes, then performing a SHA-256 hash of the resulting 20 byte secret combined with the 21 byte compressed key values in each direction, and folding the 32 byte digests once to get the required 16 byte encryption and decryption key values for AES-128.

- channel encryption key: $\text{SHA256}(\text{secret}, \text{sent-KEY}, \text{received-KEY}) / 2$
- channel decryption key: $\text{SHA256}(\text{secret}, \text{received-KEY}, \text{sent-KEY}) / 2$

Channel BODY

CS1a channel packets are designed to be very lightweight with minimum overhead for use on networks such as 802.15.4 where there is a very low MTU. The BODY is binary and defined as:

- TOKEN - 16 bytes, from the handshake, required for all channel packets
- IV - 4 bytes, incremented sequence
- INNER - the AES-128-CTR encrypted inner packet ciphertext
- HMAC - 4 bytes, the SHA-256 HMAC folded three times

The IV must be initialized to 4 random bytes during channel setup to make the individual channel packets less identifiable, and then incremented for every new channel packet created.

Using the correct channel 16 byte key with the given IV (right-zero-padded to the required 16 bytes), the channel packet can be encrypted/decrypted with AES-128-CTR.

The ciphertext then has a HMAC value calculated, with the input key being the same 16 byte key used for AES concatenated with the 4 byte IV to be a 20 byte secret key, and the body being the output of AES.

Folding

In order to minimize the over-the-wire footprint and match key sizes while using the same SHA-256 hashing algorithm that is required elsewhere, a 32-byte hash is folded once into a 16-byte value for usage as the fingerprint and as the input key for AES-128,

and the 32-byte HMAC digest is folded three times into a smaller 4-byte value for usage as the MAC on message and channel packets.

The folding is a simple XOR of the lower half bytes with the upper ones, as in this pseudocode example:

```
1  var digest = sha256("foo");
2  var folded = digest.slice(0,16);
3  for(i = 0; i < 16; i++) folded[i] = folded[i] ^
    digest[i+16];
```

The triple fold uses progressively smaller chunks of the input, as in this C code

```
1  void fold3(uint8_t in[32], uint8_t out[4])
2  {
3      uint8_t i, buf[16];
4      for(i=0;i<16;i++) buf[i] = in[i] ^ in[i+16];
5      for(i=0;i<8;i++) buf[i] ^= buf[i+8];
6      for(i=0;i<4;i++) out[i] = buf[i] ^ buf[i+4];
7  }
```

Cipher Set 2a

This profile is based on the algorithms used during the telehash v2 development process in 2013.

The required algorithms in this set are:

- **RSA 2048** - selected as a well known and highly trusted public key algorithm and size, used as the long-lived identity
- **ECC P-256** - a well known curve with many implementations, used for the ephemeral identity
- **AES 256-GCM** - trusted common implementations available

Keys

When generating an endpoint including CS2a, the public/private keypair is [RSA 2048](#)).

Binary public keys are encoded in [DER](#) as X.509 SubjectPublicKeyInfo structures. (Most cryptographic libraries support this encoding; see [RFC 5280 section 4.1](#) and [RFC 3279 section 2.3.1](#) for more details.)

The exchange ephemeral key is generated using [ECC](#) and the [p-256](#) curve, the resulting binary public key is [uncompressed](#) (65 bytes).

Message BODY

BODY bytes:

- KEYS - 256 bytes, PKCS1 [OAEP](#) (v2) RSA encrypted ciphertext of the 65 byte uncompressed ECC P-256 ephemeral public key and a 32 byte random AES key
- IV - 12 bytes, a random but unique value determined by the sender for each message
- CIPHERTEXT - [AES-256-GCM](#) encrypted inner packet and sender signature
- MAC - 16 bytes, GCM 128-bit MAC/tag digest (some GCM implementations auto-append this)

The CIPHERTEXT once deciphered contains:

- INNER - inner packet raw bytes
- SIG - 256 bytes, PKCS1 v1.5 RSA signature of the KEYS+IV+INNER

The KEYS is created by first generating a new ephemeral elliptic (ECC) public key for this exchange and a random 32 byte AES KEY, and then using the endpoint's RSA key to encrypt it to the recipient's RSA public key. The ECC keypair should be generated using the P-256 ([nistp256/secp256r1/X9.62 prime256v1](#)) curve. The ECC public key should be in the uncompressed form 65 bytes in length (ANSI X9.63 format) followed by the 32 byte AES KEY bytes. The RSA encryption should use PKCS1 OAEP (v2) padding. The result is always padded to 256 bytes by OAEP.

The SIG is calculated from the concatenation of the KEYS (256 bytes of ciphertext), IV (12 bytes), and INNER (raw packet) together and then using the sender's RSA public key (SHA-256 hash and PKCS1 v1.5 padding) to create a signature of it, resulting in a 256 byte SIG value.

The CIPHERTEXT is created by encrypting the INNER and SIG (concatenated together) using AES-256-GCM. The GCM IV/nonce parameter is the 12 byte (96-bit) IV value,

and the key parameter is the 32 byte secret value following the ECC public key from the decrypted KEYS. The "tag" size for GCM is 16 bytes (128 bits), and the additional/auth data used to compute it is the 256 KEYS bytes, resulting in the MAC value appended to the original packet.

Channel Setup

The channel encryption/decryption secret keys are generated by using [ECDH](#) with the local ephemeral ECC private key, and the remote ephemeral ECC public key. Two secret keys are generated by performing a SHA-256 hash with the derived ECDH derived secret (32 bytes) and sent/received decrypted secret keys from the KEYS value (32 bytes each):

- channel encryption key: SHA256(ecdh-secret, sent-KEY, received-KEY)
- channel decryption key: SHA256(ecdh-secret, received-KEY, sent-KEY)

Channel BODY

Channel packet binary BODY is defined as:

- TOKEN - 16 bytes, from the handshake, required for all channel packets
- IV - 12 bytes, a random but unique value determined by the sender for each message
- CHANNEL CIPHERTEXT - the AES-256-GCM encrypted channel packet
- GCM TAG/MAC - (GMAC) - 16 bytes, GCM MAC digest (some GCM libraries auto-append this with the AES-256-GCM cipher output)

The IV is 12 random bytes that must be different for every channel packet sent (incremented from a random seed) and used as the input to the GCM calculation, along with the appropriate encryption/decryption keys calculated for the exchange during setup. The tag size for GCM is 12 bytes (96 bits) and no additional/auth data is included as input to it.

Cipher Set 3a

Cipher Set 3a is based on Daniel J. Bernstein's [NaCl: Networking and Cryptography library](#). The cipher set leverages the public-key and secret-key portions of NaCl. Implementations will need to support the crypto_box, crypto_secretbox, and crypto_onetimeauth related functions.

The version of NaCl used for 3a is implemented with `crypto_box_curve25519xsalsa20poly1305`, future versions of NaCl with other configurations will be defined in different Cipher Sets.

Keys

All CS3a key pairs are generated using NaCl's `crypto_box` from the components for public-key cryptography.

Here is some example code:

```
1 | var sodium = require("sodium").api;
2 | var keys = sodium.crypto_box_keypair();
3 | console.log(keys.publicKey); // binary public key, 32
   | bytes
```

Message BODY

The BODY of a message packet is binary and defined as the following byte sections in sequential order:

- KEY - 32 bytes, the sending exchange's ephemeral public key
- NONCE - 24 bytes, randomly generated
- CIPHERTEXT - the inner packet bytes encrypted using `secretbox()` using the NONCE as the nonce and the shared secret (derived from the recipients endpoint key and the included ephemeral key) as the key
- AUTH - 16 bytes, the calculated `onetimeauth(KEY + NONCE + CIPHERTEXT, SHA256(NONCE + secret))` using the shared secret derived from both endpoint keys, the hashing is to minimize the chance that the same key input is ever used twice

Channel Setup

Channel secret keys are generated by performing a SHA-256 hash of the shared secret (`agreedKey`) and the `TOKEN` values:

- channel encryption key: $\text{SHA256}(\text{secret}, \text{sent-KEY}, \text{received-KEY}) / 2$
- channel decryption key: $\text{SHA256}(\text{secret}, \text{received-KEY}, \text{sent-KEY}) / 2$

Channel BODY

The enclosing channel packet binary is defined as the following byte sections in sequential order:

- `TOKEN` - 16 bytes, from the handshake, required for all channel packets
- `NONCE` - 24 bytes, randomly generated
- `CIPHERTEXT` - the `secretbox()` output representing the encrypted inner packet

See the [implementers guide](#) for some example code.

Packet Chunking

Sending sequential [LOB](#) packet byte arrays over streaming transports (such as TCP/TLS) requires additional framing to indicate the size of each one. Framing is also necessary to break packets into smaller pieces for low MTU transports (such as Bluetooth LE and 802.15.4 based) and to signal flow control (Serial).

Chunking is a minimalist byte-encoding technique describing how to break any packet into multiple sequential chunks and re-assemble them into packets with low overhead. There is no CRC or other consistency checks as it is only designed to carry encrypted packets with their own internal validation and be implemented as a library utility for many transports with differing needs.

Format

A packet is broken into fragments of size 1 to 255 bytes each, and each fragment is prefixed with a single byte representing its length; this combination together this is called a chunk. The sequence of one or more chunks is terminated with a zero-length terminator chunk consisting of a single null byte.

Sequential chunks received that have any size should be appended to a buffer until the zero terminator chunk, at which point the buffer should be checked for a valid packet or discarded. Any lone zero chunks with no buffer should be ignored.

Chunk size of 5:

```
packet = [0,1,2,3,4,5,6,7,8,9]; // 10 byte packet
chunk1 = [4,0,1,2,3]; // 4-byte fragments
chunk2 = [4,4,5,6,7];
chunk3 = [2,8,9];
chunk4 = [0]; // terminator
```

When using chunks for fixed frame sizes the terminator should be included in the last frame if there is space, so the last frame for the example would be `[2, 8, 9, 0]`.

Acks

At any point a zero-length chunk may be sent in response to a full incoming chunk and used by a transport for sending acknowledgement or keepalive signals. Some transports may require this to manage flow control, buffer sizes, and detect timeouts faster.

This ack mechanism should not be used to try and create a reliable transport at this level, since packets are expected to be safe to send unreliably and will internally be retransmitted when necessary.

Transport Notes

- TCP / TLS - default chunk size of 256, ensure a chunk or ack is written in response to processing one or more incoming chunks to help the sender detect timeouts faster
- BLE - default chunk size of 20 to fit into a Bluetooth LE data frame, no acks necessary
- 802.15.4 based transports - default chunk size of 120 bytes, no acks necessary (use 802.15.4 framing acks)
- Serial - use the hardware serial buffer size (64 bytes for many arduino devices) as the chunk size, and require writing an ack for every chunk read for flow control to not overflow the hardware buffer

Transport Bindings

Since telehash is designed to work over multiple network transports, there are bindings defined in this section for how to send and receive the wire packets on different ones.

UDP Transport

Direct mapping, one packet to one message, always [cloaked](#).

Local port binding is dynamic (bind to 0) unless given a specific port. Implementations should support mapping the dynamic port via NAT-PMP and UPnP when possible.

Packets larger than the MTU may be fragmented by the router, but [chunking](#) should not be used for UDP messages. packets larger than a low MTU should be dropped so that higher-level implementations can optimize the packet sizes and detect the MTU.

Timeout

A new keepalive handshake should be automatically triggered when no packets have been sent for 30 seconds in order to keep any NAT mappings active.

Discovery

UDP transports must always also listen on *:42420 with broadcast enabled and also join the multicast address group 239.42.42.42 when available.

When discovery is enabled, the announcement packet(s) should be broadcast to the local LAN subnets port 42420 and the multicast group once every 10 seconds.

Path JSON

Example [path](#) JSON for IPv4:

```
1 {  
2   "ip": "192.168.0.55",  
3   "port": 42424,  
4   "type": "udp4"
```

```
5 | }
```

Example [path](#) JSON for IPv6:

```
1 | {  
2 |     "ip": "fe80::bae8:56ff:fe43:3de4",  
3 |     "port": 42424,  
4 |     "type": "udp6"  
5 | }
```

TCP Transport

See [chunking](#) for how to encode one or more packets on a standard TCP socket. All packets must be [cloaked](#).

Local port binding is dynamic (bind to 0) unless given a specific port. Implementations should support mapping the dynamic port via NAT-PMP and UPnP when possible.

Multipath TCP

Research is ongoing on how [Multipath TCP](#) can be used to optimize a normal TCP path.

Timeout

A new keepalive handshake may be automatically triggered when no packets have been sent for 5 minutes as an extra validation that the TCP socket is still connected.

Discovery

By default a TCP transport cannot support general discovery for local networks.

When given a specific IP and port to discover, the transport should ensure that the IP is on a local subnet and may then send the announcement packet(s) directly to that IP and port. If the connection fails it may be retried once every 10 seconds, but if it succeeds and doesn't respond then no further announcements should be sent as long as it remains connected.

Path JSON

Example [path](#) JSON for IPv4:

```
1  {  
2    "ip": "192.168.0.55",  
3    "port": 42424,  
4    "type": "tcp4"  
5  }
```

Example [path](#) JSON for IPv6:

```
1  {  
2    "ip": "fe80::bae8:56ff:fe43:3de4",  
3    "port": 42424,  
4    "type": "tcp6"  
5  }
```

TLS/SSL Transport

Identical encoding to TCP (see [chunking](#)) and cloaking is only optional.

Applications may have additional requirements around the TLS identity/verification before using it as a transport.

Channels

Telehash defines a set of standard channels for implementations to support for interoperable built-in functionality.

The standard channels all use common simple names, so any application or library supporting a custom channel should use a [Collision-Resistant Name](#): a name in a namespace that enables names to be allocated in a manner such that they are highly unlikely to collide with other names.

The current list of active and draft channels can always be found on [github](#).

path - Network Path Information

Any endpoint may have multiple network interfaces, such as on a mobile device both cellular and wifi may be available simultaneously or be transitioning between them, and for a local network there may be a public IP via the gateway/NAT and an internal LAN IP. Any endpoint should always try to discover and be aware of all of the network paths it has available to send on (ipv4 and ipv6 for instance), as well as all of the paths that exist between it and any other endpoint.

An unreliable channel of "type": "path" is the mechanism used to share and test any/all network paths available. A path request may be generated automatically by any endpoint, based on an initial link, a new incoming path, a change in local network information, or application-specific signalling (e.g., a ping or connectivity validation request).

The initial request should contain an array of all of the paths the sender knows about itself, including local ones:

```
1  {
2      "c": 1,
3      "type": "path",
4      "paths": [
5          {
6              "url": "http://192.168.0.36:42424",
7              "type": "http"
8          },
9          {
```



```

10         "ip": "192.168.0.36",
11         "port": 42424,
12         "type": "udp4"
13     },
14     {
15         "ip": "fe80::bae8:56ff:fe43:3de4",
16         "port": 42424,
17         "type": "tcp6"
18     }
19 ],
20 }

```

To handle a new `path` request, a response packet must be sent back to every already known network path for the sender, as well as to any any new paths included in the request. The initiator should leave the channel open to receive any responses until the default timeout. Every path response should include a `"path": { ... }` where the value is the specific path information the response is being sent to.

```

1  {
2      "c": 1,
3      "path": {
4          "ip": "192.168.0.36",
5          "port": 42424,
6          "type": "udp4"
7      }
8  }

```

Path Types

The information about an available network transport is encoded as a JSON object that contains a `"type": "..."` to identify which type of network it describes. The current path types defined are:

- `udp4 / udp6` - **UDP** over IPv4 or IPv6, contains `ip` and `port`, may be multiple (public and private ip's)
- `tcp4 / tcp6` - **TCP** over IPv4 or IPv6, contains `ip` and `port` like UDP
- `http` - contains `url` which can be `http` or `https`, see [HTTP](#) for details
- `webrtc` - see [WebRTC](#), ideal for browsers that have only HTTP support
- `peer` - contains `hn` which is sent [peer](#) requests to provide routing assistance, optional `uri` if provided by the peer

peer - Routing Request

Any endpoint can request another to act as a [router](#) to an endpoint it is attempting to link with. These routing requests are sent in a `"type": "peer"` unreliable channel. A peer request is typically generated as the result of having a [peer path](#) for any endpoint.

A peer open request contains a `"peer": "fvif...irkq"` where the value is the hash-name the sender is trying to reach. In some cases the hashname may not be known yet and the peer request was the result of a router-generated [URI](#), included as the `"uri": "..."` value.

The `BODY` of the open request must contain an attached packet with information for the specified peer. The `BODY` will be relayed by the routing endpoint to that peer who must validate the attached packet before processing or responding to it. The validation requires application trust similarly to processing handshakes.

The recipient of the peer request is called the [router](#) and must first qualify the specified `peer` value to make sure that it is able to establish a channel with it. It must also ensure that either the sender or peer endpoints have an active link and are trusted to provide routing for.

Once validated, the [router](#) relays the original `BODY` in a [connect](#) request to the peer.

No response is ever given to the `peer` channel; it should not error based on any validation so that the requesting endpoint cannot determine any information about the state of the peer. The channel should always silently timeout and clean up.

```
1  {  
2    "c":10,  
3    "type":"peer",  
4    "peer":"fvif...irkq"  
5  }  
6  BODY: ...packet...
```

First Introductions

The first time an endpoint is attempting a link with a new peer it might not have any information other than its hashname, so it cannot send encrypted handshakes. Instead, it must attach the handshakes unencrypted, with at least one of them including the sender's

CSK. If the sender doesn't know the correct CSID it should open multiple peer channels, one with each key handshake it supports.

Automatic Bridging

When the BODY contains an encrypted [handshake](#) packet the [router](#) should determine the [ROUTING TOKEN](#) value of the handshake and create a mapping of that token to the network path by which the peer request arrived. Any subsequent encrypted channel packets received with this token should be re-delivered to that network path, providing automatic bridging.

URIs / Sessions

If a peer is willing to be a [router](#) for another peer to other unknown hashnames, it may generate a unique [URI](#) including an opaque `session` value and send it as a [handshake](#) to that peer. When the unknown hostname sends the [router](#) a handshake it must include the URI in a handshake so that the router can validate the `session` and accept correct `peer` requests from them. When the new hostname generates handshakes for the `peer` open it must also include a URI handshake again so that it can be re-verified by the destination peer.

connect - Peer Connection Request

A connect channel is only created from a [router](#) that has received and validated a [peer](#) request. The original BODY of the peer open is attached as the BODY of the `"type": "connect"` unreliable channel open packet. The original sender is included as `"peer": "uvab...hv7g"` so that the recipient can track multiple handshakes from the same source.

The recipient should parse the attached BODY as a packet and process it as [handshake](#), either encrypted or unencrypted (if the sender doesn't have the recipient's keys yet). At least one of the handshakes should be a [key](#) to guarantee the recipient can respond. If any of them are invalid the requests should be ignored and the channel will timeout silently.

When accepted, a [peer path](#) should be implicitly added to the sender's hostname via the incoming [router](#). When the processing of the attached packet results in a response handshake, it should then be delivered via a subsequent peer request via the same [router](#).

Automatic Bridging

When the incoming connect request has a `BODY` that is a validated handshake, the current network path it was received on should also be added as a network path to the `hashname` of the handshake, since the [router](#) provides automatic bridging for encrypted channel packets.

stream - Reliable Data Streams

One of the most common patterns between any two endpoints is creating streams of data. The `"type": "stream"` reliable channel is a request from one endpoint to open a stream to another.

The stream open request may contain a single additional packet attached as the `BODY`; this contains the options for this stream request, which should communicate anything the recipient app needs to determine what the stream is for and how to create it.

```
1 {  
2   "c":1,  
3   "type":"stream"  
4 }  
5 BODY: ...
```

Once accepted (and once the open packet is acknowledged), the stream may immediately begin sending data in either direction. All streamed data is attached as the binary `BODY` to every packet. The `JSON` may contain one of the following options:

- `"frag":true` - when set, the attached `BODY` should be buffered along with any other fragments in order, until a packet is received without a `frag` at which time they are all processed
- `"enc": "..."` - what encoding is the attached data, current options are: `binary` (default), `json`, and `lob`. This applies to any buffered fragments as well at the time it is processed (`enc` is not valid to be set when `frag` is true).

Any channel `end` or `err` is processed normally and has the same implications for a stream.

sock - Tunneled TCP/UDP Sockets

A channel of `"type": "sock"` is a request to create a minimal tunneled TCP or UDP socket, carrying arbitrary binary data. It may be reliable for TCP, or unreliable to carry UDP. It is designed to mirror the popular socket interfaces for the most common usage patterns, but not as an exhaustive or complete replacement.

The open packet must contain a `"sock": value` where the value is one of `connect`, `bind`, or `accept`, and usually has a `"src": {"ip": "1.2.3.4", "port": 5678}` and/or `dst` specified. An open may also contain the initial binary BODY payload to deliver if successful.

connect

Request to create a new socket to the given `dst`. Any non-error packet response indicates the connection was opened. It may have an optional `"src": { ... }` indicating the known original source ip/port when proxying/tunneling.

bind

Request for the recipient to create a listening socket and bind to it. If the `src` is included, only the `port` should be specified in it (and defaults to 0 if not) to indicate that the recipient chooses any open port to listen on. The response must include a `dst` that specifies the current `ip` and `port` bound to this channel that the recipient may use/distribute. The recipient should do whatever is necessary to ensure they are public usable addresses.

For an unreliable bind, sending and receiving UDP messages happens on the same channel using a `src` and `dst` address. The sender specifies a `dst` to originate a new UDP message from the port, and receives incoming messages with a `src` indicating where they came from.

For a reliable bind, incoming connections will individually open new `accept` channels from the recipient for each one.

accept

When the recipient has an active reliable `bind`, any new incoming connection will generate an `accept` channel and must have the `dst` of the originating `bind` along with the `src` address information.

An `accept` can also be generated for when a binding is created independently, if a hash-name is listening on a TCP or UDP port on behalf of another without using a `bind` channel. The semantics of `dst` and `src` are the same, and the `accept` channel may be reliable or unreliable.

thtp - HTTP Mapping

This is a channel mapping HTTP to telehash for the common browser user-agent and web-app patterns. Most HTTP requests can be translated directly into a `thtp` channel and back.

Packet

Any HTTP request/response is normalized into a `packet` by translating the headers into the JSON and any contents attached as the binary BODY. The headers are always lower-cased keys and string values, for requests the `:method` (upper case) and `:path` with string values are included, and for responses the `:status` with the numeric value and optional `:reason` string value are included.

The request:

```
GET / HTTP/1.1
User-Agent: curl/7.30.0
Accept: */*
```

Becomes:

```
1 | {
2 |   ":method": "GET",
3 |   ":path": "/",
4 |   "user-agent": "curl/7.30.0",
5 |   "accept": "*/*",
6 | }
```

BODY: empty

The response:

```
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Fri, 07 Mar 2014 21:33:14 GMT
Content-Type: text/html
Content-Length: 178
Connection: keep-alive
Location: http://www.fooooo.com/
```

```
<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Becomes:

```
1  {
2    ":status":301,
3    ":reason":"Moved Permanently",
4    "server":"nginx",
5    "date":"Fri, 07 Mar 2014 21:12:39 GMT",
6    "content-type":"text/html",
7    "content-length":"178",
8    "location":"http://www.fooooo.com/"
9  }
```

```
BODY:
<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

thttp channel

A new request is initiated by creating a reliable channel of type `thttp`, multiples can be created simultaneously. These channels are always in one direction, the endpoint starting the channel can only send a request packet over it, and the receiving side can only send a response packet. If the receiving endpoint needs to make requests, it can start a `thttp` channel in the other direction at any point.

The channel open packet includes all of or as much of the request packet as possible in the BODY, with subsequent packets if it needs to be fragmented and the last packet always including an `"end": true` to end the channel.

The channel is also compatible with [streams](#) such that an implementation can share the same underlying channel handler/streaming logic.

```
1  {  
2    "c":1,  
3    "seq":0,  
4    "type":"thttp",  
5    "end":true  
6  }  
7  BODY: request packet
```

The response is the same pattern, with the BODY being a response packet, continuing in subsequent packets if necessary until the `"end": true`.

```
1  {  
2    "c":1,  
3    "seq":1,  
4    "end":true  
5  }  
6  BODY: response packet
```