Internet Engineering Task Force Stefan Birgmeier
INTERNET-DRAFT I-D Unaffiliated
January 2020

Hone-DHT

Abstract

This document provides the specification of the proposed Hone-DHT protocol. Hone-DHT intends to be a versatile distributed addressing protocol to be used as building-block for applications.

Table of Content

1 Overview

The Hone-DHT aims to be a robust addressing layer that does not repeat the mistakes made by other DHT protocols. Thus, Hone-DHT by design is

- simple: ‘‘do one task and do it well’’,
- application agnostic,
- not vulnerable to the Sibyl attack,
- no attack surface for reflection-based DDoS amplification,
- easy to use for application developers,
- independent of central servers,
- extensible and customizable.

These goals are achieved by the following design choices:

- JSON formatted protocol messages,
- cryptographically generated addresses (CGA),
- handshakes required before large replies,
- no assumptions are made regarding the functioning of the upper-layer protocol or application,
- blind operation with regard to own (IP-)address (A.1.1),
- flexibility with respect to the choice of cryptographic primitives (A.1.2).

2 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Any mention of ‘‘base64’’ throughout this RFC refers to RFC 4648, Section 3.5 (5) ‘‘Base 64 Encoding with URL and Filename Safe Alphabet’’.

3 Protocol Overview

3.1 Cryptographic Design

Each node has a unique 256-bit address, called the ‘‘Node-ID’’. This address is derived from a node’s public key by applying the SHA-256 hash to its JSON representation. This key is called the node’s ‘‘main key’’. There is a second type of key which is used to sign certain types of messages. This key is called the ‘‘current key’’ and is itself signed by the node’s main key (A.1.3).

3.2 Distance Metric

Hone-DHT uses the XOR-distance. For this purpose, the Node-ID is regarded as 256 bit integer:

```
          +---------+---------+-...-+--------+
Node-ID = | byte 31 | byte 30 | ... | byte 0 |
          +---------+---------+-...-+--------+
```

```
INT(Node-ID) = (byte 31) * 256^31 + (byte 30) * 256^30 + ... + (byte 0) * 256^0
```

The bytes are interpreted as unsigned 8 bit integers. The distance between two nodes is small if the most significant bits in the most significant bytes are zero. In base64 representation, the least significant bits are converted first, thus a small ID (or distance) is indicated by the base64 representation ending in ’A’s.

In this specification, the distance is denoted using d while the number of leading zeros (i.e. number of consecutive most significant bits which are zero) of d is denoted by d_{lz}.

3.3 Protocol Field Names

The following protocol field names exist (not all appear in all messages):

field name	abbreviation
current key	ck
lookup-id	id
message type	m
main key	mk
node list	nl
next request-id	nrid
namespace	ns
request-id	rqid
source node-id	src

Table 1: List of protocol field names and their abbreviations.

Notably, a destination field is missing - each node needs to be uniquely addressable using lower layer protocols. It is therefore not possible to run multiple nodes on the same socket (protocol, ip, port), for example. An advantage of this is that no Node-IDs need to be known for bootstrapping.

3.4 Message Types

The protocol consists of 15 types of messages. These provide functionality to ping other nodes, find other nodes, check if a node participates in a certain namespace and to get other node's public main and current public keys. The message types are:

message type	abbreviation
ping	pi
pong	po
find-nodes	fg
find-nodes-syn	fs
find-nodes-ack	fa
find-nodes-reply	fr
get-main-key	mg
get-main-key-syn	ms
get-main-key-ack	ma
get-main-key-reply	mr
get-current-key	cg
get-current-key-syn	cs
get-current-key-ack	ca
get-current-key-reply	cr
unknown-namespace	xn

Table 2: List of message types and their abbreviations.

The ‘‘find-nodes’’ message is abbreviated using ‘‘fg’’ to make parsing

easier: all three-way messages for information retrieval (i.e. ‘‘find-nodes’’, ‘‘get-main-key’’ and ‘‘get-current-key’’) use the same suffix for the same message type. These suffixes are:

message type	synonym	letter
initiating message	get	g
request for ack	syn	s
response ack	ack	a
response with information	reply	r

Table 3: List of message suffixes.

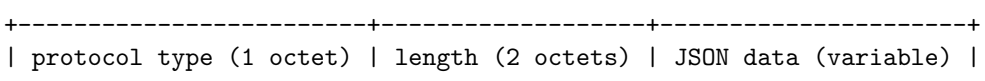
To avoid ambiguity with protocol field names, the ‘‘find-nodes’’ abbreviation uses ‘‘f’’ as initial character, thus avoiding mapping ‘‘find-nodes-syn’’ to ‘‘ns’’, which is used to indicate the namespace.

3.5 Wire Format

Any message received and processed by a Hone-DHT node is either a Hone-DHT message or a namespace-specific message. The two protocols are distinguished by the first byte of the message. Possible values for this byte are

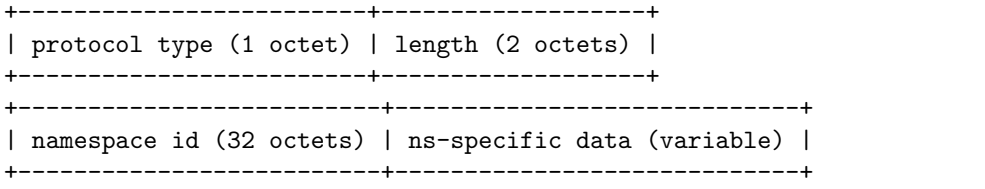
Value	Protocol
0	Hone-DHT
1	namespace-specific
2-255	reserved

A Hone-DHT message looks as follows:



The length field is in network byte order and specifies the length of the JSON data in octets.

A namespace specific message looks as follows:



The length field is in network byte order and specifies the length of the ns-specific data.

3.6 Message Fields

The following table shows which fields appear in which messages:

message type	m	src	rqid	nrid	ns	id	nl	mk	ck
ping	y	y	y		o				
pong	y	y	y		d				
find-nodes	y	y	y		y	y			
find-nodes-syn	y	y	y	y					
find-nodes-ack	y	y	y	y					
find-nodes-reply	y	y	y		y		y		
get-main-key	y	y	y						
get-main-key-syn	y	y	y	y					
get-main-key-ack	y	y	y	y					
get-main-key-reply	y	y	y					y	
get-current-key	y	y	y						
get-current-key-syn	y	y	y	y					
get-current-key-ack	y	y	y	y					
get-current-key-reply	y	y	y						y
unknown-namespace	y	y	y		y				

Table 4: Message-type to field association.

In the Table 4, fields marked with

y MUST be present,

(empty) MUST NOT be present,

o MAY be present depending on desired behavior,

d NEEDS to be present or absent, depending on other message's optional behavior.

3.7 Message Signatures

Whenever it makes sense, messages are signed. A message is signed by inserting the message into the ''data'' field of signed data (c.f. 6.7). Unsigned messages are still encapsulated within a JSON object with a single ''data'' field, omitting the ''sig'' field, to simplify implementations. The following table shows which messages MUST be signed:

message type	signed	initiating	reasoning
ping		y	(A.1.4)
pong	y		
find-nodes		y	(A.1.4)
find-nodes-syn	y		
find-nodes-ack	y		
find-nodes-reply	y		
get-main-key		y	
get-main-key-syn			
get-main-key-ack			
get-main-key-reply			(A.1.5)
get-current-key		y	
get-current-key-syn			
get-current-key-ack			
get-current-key-reply	y		(A.1.6)
unknown-namespace	y		

Table 5: Message signature presence.

Nodes sending any of the messages marked as ‘‘initiating’’ in table 5 MUST verify the signatures of all signed messages received which are related to the initiating message.

Nodes SHOULD verify the signatures of messages they receive if the corresponding public key is cached (A.1.7).

State information MUST NOT be updated in response to unsigned messages or messages with invalid or unchecked signature.

3.8 Message Validation and SYN Cookies

For all except initiating messages, information is already known about the remote node. This information includes the remote Node-ID and the remote ConnSpec. However, the target Node-ID might sometimes not be known when sending initiating messages.

Based on available knowledge and message sent, additional checks SHOULD be performed for replies:

- The ‘‘m’’ field’s value MUST be as expected.
- The ‘‘rqid’’ field’s value MUST be as expected.
- The ‘‘src’’ field’s value MUST be as expected.
- The ConnSpec of the sender MUST be equal to the expected ConnSpec.

SYN Cookies MAY be used instead of keeping lists of request IDs with associated information on expected messages. They SHOULD NOT be used for initiating messages, follow-ups to initiating messages (A.1.8)

or messages where no reply is expected. Messages that qualify for the usage of SYN-cookies are ‘‘find-nodes-syn’’, ‘‘get-main-key-syn’’ and ‘‘get-current-key-syn’’. The additional checks specified above SHOULD be included in the generation of the SYN Cookies. SYN Cookies MUST contain an element preventing replay-attacks such that a large number (in a cryptographic sense) of identical incoming messages from the same ConnSpec result in different SYN Cookies. If an implementation cannot ensure sufficient SYN Cookie uniqueness, incoming signed messages MUST NOT affect local node state.

One way to avoid massive replay attacks is the use of a leaky counting Bloom filter. Special care must be taken if the implementation intends to support detection of retransmitted messages (A.1.9).

3.9 Retransmissions

If a node expects a reply to a message it sent and the reply does not arrive within a reasonable time frame, it MAY choose to retransmit its last message. A message MAY be retransmitted up to three times (a total of four times when counting the initial transmission). It MUST NOT be retransmitted once a valid reply is received. Retransmissions of non-initiating messages MUST be identical to the originally transmitted message.

Regular implementations cannot recognize retransmissions: after receiving a message with a certain rqid, the rqid is removed from the list of expected messages. In this case, retransmissions only remedy packet loss in the uplink.

Extended implementations might provide mechanisms to recognize retransmitted messages. Answers to retransmitted messages MUST be identical to the original answer. A node MUST NOT reply to more than three retransmissions.

The reasonable time frame within which a reply should arrive must be limited. Apart from this, implementations are free to decide on mechanisms to optimize timeouts. A statistical approach using the known round-trip times of each node, as well as the worst-case round-trip times (useful for communication with unknown nodes) is recommended. The limit must ensure that such statistical algorithms cannot be abused to make a node wait for replies too long, exhausting its resources.

4 Messages

Only field name abbreviations are used for the wire-formatted messages. Each message's fields MUST be ordered in ascending fashion (A.2.2). The ordering is performed by comparing each byte of the abbreviation, regarded as signed 8-bit integer. If two names are equal up to the shorter of their lengths, the shorter one is regarded as smaller than the longer one. The field names in Table 1 are properly ordered.

When a node receives a message, it SHOULD reply. A node MAY omit a reply if

- it is in the process of shutting down or
- it is overloaded or
- it suspects abuse (e.g. DDOS, exploiting of bugs or weaknesses in a cryptographic system) or
- it suspects invalid implementation of the sending node (e.g. pings sent too often).

A node MUST NOT reply if

- the message is not an initiating message and not related to any initiating message sent by the receiving node and not related to any previously received initiating message for which a reply was sent or
- the message's format is incorrect (e.g. missing signature, containing whitespace, containing invalid characters, invalid ordering of fields) or
- the message's signature is incorrect.

The ``nrid'' field sets the next request ID and MUST be different from the current request ID and MUST be different from all previously used request IDs with cryptographic certainty. Any message that is a reply to a message with an ``nrid'' field MUST use the ``nrid'' as ``rqid''. Message relations are discovered by comparing the ``rqid'' field to the expected request ID, which is the ``nrid'' sent in the previous message (A.2.1).

A node's current key MUST NOT change during a message sequence. If a node receives a message which indicates that it is signed using a different current key than a previous message in the same sequence, the message MUST be ignored (A.2.3). A message sequence is an exchange of related messages, such as ping - pong or find-nodes - find-nodes-syn - find-nodes-ack - find-nodes-reply, which are linked by their request IDs.

4.1 Ping

Pings are used to verify node presence. If a ping contains the ‘‘ns’’ field, it is namespace-specific. Upon receiving a ping, a node SHOULD answer with one of

pong Signal the pinging node that the pinged node is reachable and active in the specified namespace, if set.

unknown-namespace Signal the pinging node that the pinged node is not active in the specified namespace. This reply is only valid if the namespace was set.

An ‘‘ns’’ field present in a ping does not indicate that the sending node is active in the specified namespace. It is therefore possible to check if a node is active in a certain namespace without the pinging node being active in the namespace itself.

4.2 Find-Nodes

The ‘‘find-nodes’’ message is used for node discovery. The supplied ‘‘id’’ field specifies the search target. A node receiving a ‘‘find-nodes’’ request SHOULD answer with one of

find-nodes-syn Request verification that the sending node is reachable at the address specified in the transport protocol.

unknown-namespace Signal that the receiving node is not active in the specified namespace.

Upon receiving a find-nodes-ack, a node SHOULD reply with a find-nodes-reply. It

- MUST NOT return the requesting node in its node list reply,
- MUST NOT return itself in the node list reply,
- MUST NOT return more than 20 nodes at once,
- SHOULD NOT return less than 20 nodes unless it does not know at least 20 nodes in the requested namespace,
- MUST NOT return any nodes without a single ConnSpec,
- MUST NOT list more than 10 ConnSpecs for any returned node,
- MUST return an empty node list only if it does not know any other nodes in the requested namespace,
- MUST NOT return any nodes where it did not verify the main and current public key, furthermore

- the reply's node list SHOULD only contain nodes that can be expected to be reachable by the requesting node.

4.3 Get-Main-Key

This message is used to get a node's main public key. The get-main-key-reply is not signed, however it is easy to verify the main key.

- The SHA-256 hash of the public key MUST match the Node-ID. A node whose main key is invalid (e.g. invalid format, expired or not yet valid) MUST be deleted from the local state.
- The key does not need a ''validto'' field. If it is absent, the node is valid indefinitely. If a ''validto'' field is present, the node MUST be removed from the local state after the key is expired.
- The key does not need a ''validfrom'' field. If it is absent, the node is valid immediately. If a ''validfrom'' field is present, the node MUST NOT be added to the local state before the key becomes valid.
- If both ''validfrom'' and ''validto'' fields are present, their difference MUST be at lest 300000 (5 minutes) and ''validfrom'' MUST be smaller than ''validto''.
- If the difference between ''validfrom'' and ''validto'' is less than 10800000 (3 hours), the node SHOULD NOT be added to the local state.
- Similarly, if the difference between the current unix timestamp with millisecond precision and the ''validto'' field is less than 10800000 (3 hours), the node SHOULD NOT be added to the local state.
- The (unencoded) key ID MUST be ''mk'' (encoded: ''bWs'').
- The purpose field MUST be present and contain the sole purpose ''mk''.
- The key MUST NOT be published outside its validity period and the node MUST NOT be used while its key is not valid.

4.4 Get-Current-Key

This message is used to get a node's current public key. The current key reply is signed to prevent spoofing with old current public keys.

- Both ''validfrom'' and ''validto'' fields MUST be set.

- The current key's ‘‘validfrom’’ field MUST NOT be smaller than the main node key's ‘‘validfrom’’ field. If the main node key's ‘‘validfrom’’ field is not set, its value is assumed to be negative infinity for the purposes of this rule.
- A new current key's ‘‘validfrom’’ field MUST be larger than the last current key's ‘‘validfrom’’.
- The current key's ‘‘validto’’ field MUST NOT be larger than the main node key's ‘‘validto’’ field. If the main node key's ‘‘validto’’ field is not set, its value is assumed to be infinity for the purposes of this rule.
- The difference of the current key's ‘‘validfrom’’ and ‘‘validto’’ fields MUST be at least 300000 (5 minutes). The current key's ‘‘validfrom’’ field MUST be smaller than its ‘‘validto’’ field.
- The current key's non-encoded ID SHOULD be at most 3 bytes long.
- The current key's ID MUST change every time a new current key is generated. The ID SHOULD NOT repeat within 30 minutes.
- The current key's purpose field MUST be present and contain the sole purpose ‘‘ck’’.
- The current key in the current key reply MUST be signed by the main node key. The signature MUST be computed only once for each current key and remain identical in every reply.
- The current key MUST NOT be published outside its validity period.

5 Node State

Each node keeps a state containing information about other nodes in the network. At least 20 nodes with $d_{\mathrm{lz}} < d_{\mathrm{lz}}^{\max}$ should be cached, where $d_{\mathrm{lz}}^{\max}$ is the distance with the largest number of leading zeros, defined by the closest known node.

Each node MUST take steps to keep the node state updated by removing nodes that become unreachable, whose keys expire or which misbehave in other ways, including, but not limited to, misbehavior as specified in this RFC. The state update procedures MUST be implemented in a way that limits network traffic. Any time intervals between update procedures MUST be randomized, possibly around a fixed mean value. Unsigned messages or messages whose originality cannot be verified with a cryptographic level of certainty MUST NOT influence the node state.

A node state MUST contain at least the following information:

- A list of known nodes.
- For each node,

    { the Node-ID,

    { the node's main public key and its validity period, if set,

    { the node's current public key and its validity period,

    { a limited number of ConnSpecs, but at least one,

    { the namespaces it is active in,

    { the time when the last signed message was received.
- A list of namespaces the local node is active in.

A node state SHOULD contain the following information:

- For each node,

    { the round-trip time, possibly per ConnSpec.

A node that is not known to be active in any namespace MUST be removed from the node state. The node state NEED NOT keep track of other nodes' current key IDs or enforce the uniqueness of other nodes' current key IDs within a 30 minute window. A node SHOULD store a reasonable amount of its state in permanent storage to minimize network traffic at startup.

6 Object Definitions

Hone uses various ‘‘Objects’’ in its messages, such as descriptions of nodes, keys, identifiers etc. Their representation and possible values are listed below. Whenever examples of a JSON representation are given, note that the protocol does not allow for whitespace (e.g. newline) or non-ASCII or non-printable characters. Any whitespace, indentation etc. is thus added to improve readability. Furthermore, examples might not be complete. Missing objects are replaced by an ellipsis (‘‘...’’).

6.1 Identifiers

In Hone, Identifier Objects (IDs) are used as

- Node Identifier,
- message and next message Identifier,
- Namespace Identifier,
- Lookup Identifier.

Identifiers are 32 bytes (256 Bit) long and encoded with base64, without any trailing ’=’. They can be randomly generated or derived from a cryptographic algorithm. If they are generated according to some algorithm, they SHOULD still be indistinguishable from randomly generated Identifiers. Note that the base64 variant used is base64url.

Example 1:

Pfq5qRDL5S5xETEweRDHxKZ9z8fjWlZETHtgsEI09So

Example 2:

gpTHjOerwYujuqO_ZFH8jqjzAi2EeYeLE1sQPsh75k0

6.2 Connection Specification

A Connection Specification (ConnSpec) describes an endpoint at which a node is reachable. It is encoded as JSON object and MUST have a field ‘‘pr’’ of string type which specifies the protocol (and thus, the format) of the Connection Specification. At the moment, UDP and TCP are defined protocols. Implementations intended for use on the Internet MUST be able to handle both IPv4 and IPv6 addresses. Both

TCP and UDP ConnSpecs are defined by the fields ‘‘a’’, which contains the IP address as string, and ‘‘p’’, which contains the port as integer. IPv6 addresses SHOULD be properly abbreviated.

protocol	‘‘pr’’ field value
TCP/IP	tcp
UDP/IP	udp

Table 6: List of ConnSpec protocols.

Example 1:

```
{
    "a":"1.2.3.4",
    "p":22222,
    "pr":"udp"
}
```

Example 2:

```
{
    "a":"2001::1:2",
    "p":22223,
    "pr":"tcp"
}
```

6.3 ConnSpec List

A ConnSpec List is an array of ConnSpecs. In the Hone protocol it appears in the Find-Nodes-Reply message. It is considered courteous to sort the ConnSpec List such that the most likely reachable addresses (e.g. determined as most-recently-used) appear before less likely reachable addresses. Since this cannot be verified, the receiving node MUST NOT rely on the order of addresses in the ConnSpec List.

Example:

```
[
    {
        "a":"2001::1:2",
        "p":22223,
        "pr":"tcp"
    },{
        "a":"1.2.3.4",
        "p":22222,
        "pr":"udp"
    }
]
```

6.4 Node

In the Hone protocol, the Node object appears in the Find-Nodes-Reply message. It contains two fields: the ‘‘id’’ specifies the Node-ID in base64 format while the ‘‘cs’’ field contains the list of ConnSpecs.

Example:

```
{
    "cs":[...],
    "id":"y11WUQFaH4Lsrrfhq8Wqp4HBkKAGqwmQkIyTLqc39Kk"
}
```

6.5 Node List

In the Hone protocol, the Node List object appears in the Find-Nodes-Reply message. It is an unordered array of Node objects.

Example:

```
[
    {
        "cs":[...],
        "id":"y11WUQFaH4Lsrrfhq8Wqp4HBkKAGqwmQkIyTLqc39Kk"
    },
    {
        "cs":[...],
        "id":"zNgh-UwX8CB3FoO1sFCxzN6gCUnU-El3VmpIZ-q3oaM"
    }
]
```

6.6 Public Key

The public key is represented as JSON object (A.3.1) with the following fields:

csys String specifying the cryptographic system, e.g. ‘‘ed25519’’.

key The key material. Binary values MUST be encoded using base64 without padding (’=’) or newlines. A suitable JSON format (e.g. string or object) should be chosen when a new algorithm is implemented.

id A value between 1 and 32 bytes long (before encoding), base64 encoded. All simultaneously valid keys in use by a node MUST have distinct key IDs regardless of cryptographic system. Note that node keys (main and current) have more restrictive key id specifications.

vf (validfrom) Integer, milliseconds since the Unix Epoch. Used to specify when the public key becomes valid (the key becomes valid at the start of the specified millisecond). If omitted, key validity does not have a lower limit. The integer MUST be representable using a signed 64bit integer. (A.3.2)

vt (validto) Integer, milliseconds since the Unix Epoch. Used to specify the end of the key's validity (the key becomes invalid at the start of the specified millisecond). If omitted, key validity does not have an upper limit. The integer MUST be representable using a signed 64bit integer. (A.3.2)

pp (purposes) A JSON array of strings describing valid usage of the key. Hone knows two purposes and MUST NOT accept any keys containing any other than exactly one of the following purposes:

mk Node main key

ck Node current key

Example 1 (indefinitely valid node main public key):

```
{
        "csys":"ed25519",
        "key":"oA8241QzrLfrwFhn4qxv-lO0kOIAC1Hrmpf2S3m7PXo",
        "id":"bWs",
        "pp":["mk"]
}
```

Example 2 (node main public key with half-infinite validity period):

```
{
        "csys":"ed25519",
        "key":"oA8241QzrLfrwFhn4qxv-lO0kOIAC1Hrmpf2S3m7PXo",
        "id":"bWs",
        "vf":1573419437000,
        "pp":["mk"]
}
```

Example 3 (node main public key with finite validity period of 31 days):

```
{
        "csys":"ed25519",
        "key":"oA8241QzrLfrwFhn4qxv-lO0kOIAC1Hrmpf2S3m7PXo",
        "id":"bWs",
        "vf":1573419437000,
        "vt":1576097837000,
        "pp":["mk"]
}
```

Example 4 (node current public key with 14 day validity period):

```
{
        "csys":"ed25519",
        "key":"cCWjoAlR1X3oVISFvHw8eSz3eBRRJi-bAzu5mDvXHuE",
        "id":"FAEA",
        "vf":1573419137000,
        "vt":1574629037000,
        "pp":["ck"]
}
```

6.7 Signed Data

Signed data is represented by a JSON (A.3.1) object with the following fields:

data A JSON object containing the Hone message.

sig A JSON object containing the signature.

The ‘‘data’’ JSON object contains the Hone message or other signed data (the current node key is signed in the Get-Current-Key-Reply message). The ‘‘sig’’ JSON object consists of fields which result from the signature process and fields that specify how the signature was created and which key was used for signing. The signature is created by signing the complete Signed Data structure, omitting any fields that contain data which depends on the result of the signing process. The ‘‘sig’’ JSON object MUST contain the following fields

keyid Base64 encoded key ID of the public key which can be used to verify the signature.

Example:

Data supplied to signature algorithm:

```
{
        "data":{
                "csys":"ed25519",
                "key":"cCWjoAlR1X3oVISFvHw8eSz3eBRRJi-bAzu5mDvXHuE",
                "id":"FAEA",
                "vf":1573419137000,
                "vt":1574629037000,
                "pp":["ck"]
        },
        "sig":{
                "keyid":"bWs"
        }
}
```

Signed Data:

```
{
        "data":{
                "csys":"ed25519",
                "key":"cCWjoAlR1X3oVISFvHw8eSz3eBRRJi-bAzu5mDvXHuE",
                "id":"FAEA",
                "vf":1573419137000,
                "vt":1574629037000,
                "pp":["ck"]
        },
        "sig":{
                "keyid":"bWs",
                "sig":"QzTi-O9gGnACPo9kJM35ppdqZ5T2yrJIqEU_XcagCgU
                       x8G2dDVGMXXLVq0CcxrzE-JXe_yJS8ho65_BQ99xXAQ"
        }
}
```

A Reasoning

A.1 Protocol Overview

A.1.1 Blind operation with regard to own (IP)-address

Many DHT specifications silently assume that the own address is known. However, in many cases it is difficult to detect the public IP address (e.g. carrier NAT, home firewalls). Those DHTs then use external services to discover their IP address. These services are centralized and thus represent a single point of failure. They can also be used to centrally detect usage of the DHT. Finally, their answers are often trusted without any checks. More problems arise if the user is mobile or their public address changes for any other reason.

Hone-DHT nodes do not need to know their own public address - it is sufficient if the targets of their messages know it. Furthermore, it makes bootstrapping easier since node IDs do not need to be known, it is sufficient to know an address. The only disadvantage is that it is not possible to run multiple nodes on the same socket (protocol, address, port).

A.1.2 Flexibility towards cryptographic primitives

New protocols are often designed around a single cryptographic primitive. This is in contrast to older protocols (such as TLS), which offer a wide variety of cryptographic primitives. The argument is that the flexibility of protocols like TLS adds little to security while often opening up avenues for attacks (e.g. cipher downgrades, protocol downgrades). Hone-DHT tries to pursue a middle ground by specifying only a single cryptographic primitive in its main RFC (namely Ed25519), while making the protocol flexible enough to allow for different cryptographic systems. As long as Ed25519 remains secure, there should be no reason to introduce additional algorithms. Many cryptographic algorithms have not aged well, however. It should not be expected that Ed25519 lasts forever. To avoid situations like in I2C (which moved from RSA to Ed25519), which involved a network split, the protocol is flexible. Furthermore, the slightly extended key format enables key validity periods, which however adds the dependence on a secure time source (which, as of now, does not exist - unless central services are trusted). The dependence on secure time sources can be mostly avoided by using main node keys with infinite validity. An approximate time can be used for validity checks of current node keys.

A.1.3 Main and current key

This choice to use two different keys was made to protect the main key, which can have a long lifetime. While the short-lived current key is used to sign messages (which can be forced by an attacker flooding the node with requests), the main key only signs each current key. Thus it is not possible for an attacker to harvest large numbers of signatures by the main key.

Future Hone-DHT extensions may also make it possible for multiple nodes to used the same current key, thus minimizing the update effort. Furthermore, apart from requiring twice the memory consumption, the the overhead of keeping current keys updated is limited due to their relatively long lifetime. The lifetime of the current key is suggested to be between one week and one year.

A.1.4 Missing signature of initiating messages

There are two reasons to sign data in Hone: establishing a relationship between main and current public keys and cryptographically proving that transmitted data is up to date and originates from a certain node. Initiating messages contain no information apart from node presence. Since initiating messages can be replayed without this being detected (even if they were signed), they cannot be used to update the local node state. Verifying the signature of such a message does not provide any benefit: instead of forging a message and not being able to correctly sign it, an attacker could simply replay a previously recorded initiating message.

A.1.5 Missing signature of get-main-key-reply

The purpose of signatures is to certify that a message originates from a certain node. The message ‘‘get-main-key-reply’’ contains the main key as payload. It is easy to verify that the main key matches the node by computing its SHA256-hash. It is of course possible that the reply did not originate at the node, however (as long as SHA256 is not broken) the key is the requested main node key with cryptographically high probability. For blind key requests (i.e. where the ID of the node at a certain ConnSpec is not known) it is of course possible for a hostile party to intercept the network traffic and substitute their own node for the one actually running at the ConnSpec. Hone-DHT does not solve the fundamental problem of identity verification - to be sure that one is connected to a certain third party, it is necessary to compare their public key in person, or over a pre-existing secure channel (the chicken-and-egg problem).

A.1.6 Signature of get-current-key-reply

Note that the signature on the get-current-key-reply message is created using the same current key that is shipped with this message. The signature of the message is not intended to provide authenticity of the message. Instead, it is intended to show possession of the current key. This could also be done by merely signing the request ID which was set using the ‘‘nrid’’ field in the ‘‘get-current-key-ack’’ message. It seems sensible to use a signature of a message where only a relatively small part is controlled by the other party.

A.1.7 Signature verification of other messages

This section concerns signature verification of messages that are not related to initiating messages of the verifying node. In these cases, the verifying node does not have any interest in information contained on the messages, thus does not initiate changes to its cache. Furthermore, messages can only be verified if the current key is already known. If it is not known, fetching it would violate a fundamental principle of Hone-DHT, i.e. not doing any extra work, especially if it is initiated by a third party.

There is another problem with checking the signatures of messages which were not initiated by the verifying node: it leaks the state of the local cache (which is not really secret). Replying to messages with an invalid signature is suboptimal, however. This is an open problem.

A.1.8 No SYN-cookies for initiating messages

A node is expected to send a limited number of messages. The number of anticipated replies to initiating messages and follow-up messages to initiating messages should be limited. It can therefore be expected that a node keeps a full list of expected request IDs. This list can be searched efficiently in $\log(N)$ time.

A.1.9 SYN-cookies for non-initiating messages

Non-initiating messages are messages sent by another node. The local node typically does not intend to obtain any information from these messages and thus does not update its cache. Cryptographic origin verification is thus of limited value. However, a node is still required to avoid being used as mirror of a reflection attack. SYN-cookies open up reflection attacks because there is no way to tell if they have been reused: even if the generated cookie includes information such as the expected method, source node ID, these are easy to spoof. In order to reuse a SYN-cookie, the attacker needs to know it first, however. Thus, it needs to be sent to the attacker at some point.

It is therefore useful to also include the expected sender ConnSpec in the generation of the SYN-cookie. This then strictly enforces zero mobility during the message exchange (i.e. the node cannot change address). As additional precaution, it might be advisable to use a leaky bloom filter or similar methods to detect cookie-reuse attempts.

A.2 Messages

A.2.1 Next request ID

The next request ID field (‘‘nrid’’) is used to chain messages cryptographically. It provides replay protection because each reply contains data (namely the request ID as requested in the ‘‘nrid’’ field) that cannot be predicted and is never reused. A simple implementation simply keeps a list of expected request IDs (i.e. the contents of the ‘‘nrid’’ field of messages it sent) and matches incoming messages against this list (which can be done in $\log(N)$ time). This, together with signatures ensures replay protection.

A.2.2 Message field ordering

This provides cryptographic rigidity, i.e. makes it harder to construct a message matching a different message’s signature.

A.2.3 Constant current key during message sequence

The current key must not change during a message sequence. For example, when sending a find-nodes and subsequently receiving a signed find-nodes-syn, it must be signed by the same current key as the find-nodes-reply which is received following the find-nodes-ack. Otherwise, a node might be forced to do a key lookup twice during a message sequence. Since message-timeouts are in the second-range, this mechanism does not force a node to keep an old current key around for an inconvenient amount of time. Generally, current keys should be provisioned such that their use starts well into the validity period (to allow for inaccurate clocks at remote nodes). Furthermore, use should end well before the end of the validity period, allowing the key to be used to sign message sequences associated with the old key without using the key after the validity period, and allowing for inaccurate clocks.

A.3 Object Definitions

A.3.1 Cryptographic JSON objects

The JSON format for keys and signatures was created before the Web Cryptography Api (WCA) existed (Hone-DHT was conceived in 2013) and

thus it is currently incompatible. While support for WCA formats can easily be added, it is unlikely to completely replace the custom format completely. The reasoning behind this is that WCA formats still rely on OID-numbers, which do not exist for all (especially newer) cryptographic algorithms. Furthermore, at time of writing, WCA does not provide support for *any* safe elliptic curve cryptography algorithms.

It should be emphasized that (we think that) the custom JSON format is in no way less secure than the WCA format. It is even simpler, since it only allows for fields used by Hone-DHT.

A.3.2 Validfrom and Validto fields

These fields have millisecond precision. This enables the validfrom field to be used as a nonce when one desires to generate keys that hash to a particularly ‘‘nice’’ value. Such a key is mainly useful for debugging. It is counterproductive to use these keys in production systems, since human observers might start trusting nodes based on the readable part of their ID (which can easily be spoofed).

The fields specify the start (‘‘top’’) of a millisecond, thus a key that uses the same values for both fields is never valid and the validity period can be computed using subtraction. Note that the time period covered by 64bit numbers using millisecond precision amounts to over 584 million years, of which of course about 292 million are in the past. This still leaves engineers a convenient number of millennia to come up with something better.