Contents

List of Tables

List of Figures

1 Abstract

This document is meant to describe a peer-to-peer networking protocol that can be reasonably implemented in any popular language. Its principle goals are

1. Must work with nodes which can only initiate connections (done)
    1. Caveat: at least one node must be able to accept connections (done)
2. Must be capable of network-wide broadcasts (done)
    1. Must scale better than $O(n^2)$ (done)
    2. Lag factor (compared to hub-and-spoke) must be < O(n) (done)
3. Must be capable of broadcasting only to direct peers (done)
4. Must be capable of sending messages to specific nodes (done)
    1. And be able to optionally encrypt it (opt-in or opt-out) (done)
5. Should have public keys as address (done)
6. Should be able to support a Kademlia[1]-like distributed hash table (done)
    1. This table should be able to have locks
    2. This table should be able to support atomic changes
    3. This table should be able to support diff-based changes

2 Notes

While this document will make references to object diagrams, please be aware that we are not dictating how you must implement things. If a different implementation can achieve the same results, then by all means use it, especially if it's simpler or more elegant.

Also, this document will largely be written with Python in mind, partly because it reads like pseudocode, and partly because I am most comfortable with it. I will use type annotations as a guide for static languages.

3 Constants

3.1 Network Constants

These are the set of constants which manage network topology. They determine things like how many peers one can have, or the number of bits in your address space. Explanations will be given when these numbers are non-arbitrary.

- k: Kademlia[2]'s replication parameter (max size of k-bucket, # of `STORE` calls)
- α: Kademlia's concurrency parameter (number of parallel lookups)
- τ: Kademlia's address size (number of bits to consider per address/hash)
- β: The size of an extended address (bit length of public key)
- ℓ: The limit on a node's self-initiated connections (at most $k\tau + 2k - \lceil k \times \log_2(k+1) \rceil$

3.2 Opcodes

These are the values of the various opcodes used in this project. While their values are arbitrary, their ranges are chosen to take the smallest space possible when serialized.

- `ACK`: 0
- `NACK`: 1
- `PING`: 2
- `SET_CONNECTION_OPT`: 3
- `ANNOUNCE`: 4
- `CHANGE_KEY`: 5
- `SHOUT`: 6
- `SPEAK`: 7
- `WHISPER`: 8

[1] https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf
[2] https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

- FIND_NODE: 9
- FIND_VALUE: 10
- STORE: 11
- RESERVED: 12-14
- CUSTOM: 15

3.3 Connection Options and Settings

These are the values of the various connection options used in this project. In the lexicon of this paper, ”option” will refer to a key, while ”setting” will refer to a value. So for the compression option, you can have a setting `zlib`. While their values are arbitrary, their ranges are chosen to take the smallest space possible when serialized.

3.3.1 Compression

This option is used to set what compression methods are available. The default is that none are available. If the method is not supported by the peer, they will return a NACK.

Option: 0

Settings:

- none: 0 (default)
- bz2: 1
- gzip: 2
- lzma: 3
- zlib: 4
- snappy: 5
- reserved: 6-7

3.3.2 Preferred Compression

This option is used if you have a preferred compression method. The default is to let your peer decide. If the method is not supported by the peer, they will return a NACK.

Option: 1

Settings:

- none: 0
- bz2: 1
- gzip: 2
- lzma: 3
- zlib: 4
- snappy: 5
- reserved: 6-7

3.3.3 Subnet

This option is used to confirm that you belong to the same network. It compares your network constants and a description of the network. If any of these differs it returns a NACK. Upon a NACK for this, sent or received, you should disconnect.

Option: 2

Setting: [k, α, τ, β, ℓ, <network description>]

4 Message Format

4.1 Segmentation

Messages in this protocol can—and should—be batched together before sending. Because of this, we need to define segments.

4.1.1 Transmission Header

The transmission header consists of 6 bytes. The first 2 bytes contains the option section. It consists of a bitmap describing how the transmission is packed. This table is shown below. The other 4 bytes contain a big endian, unsigned integer which says how long the rest of the transmission will be.

Table 1: Transmission Header Bitwise Layout

Bits	Meaning
0-12	Reserved
13-15	Compression method (as defined in network settings)
16-47	Length of remaining transmission

4.1.2 Message Header

The message header consists of 114 + (β ÷ 4) bytes described in the below table. (38 of this comes from metadata added by our protocol, 76 from DER overhead, and 2β ÷ 8 from the keys themselves.)

The signature is applied to all parts of the message that come after it. In other words, it is based on everything from bit 256 onwards, including the payload.

Table 2: Message Header Bitwise Layout

Bits	Meaning
0-255	RSA signature (SHA-256, PSS padding)
256-287	Length of message payload
288-291	Operation (as defined in RPCs)
292-302	Reserved
303	Indicates whether the message is encrypted
304-(607+β)	From public key (DER format)
(608+β)-(911+2β)	To public key (DER format)

4.1.2.1 Isn’t that a little large?

Yes. But there are some reasonable counterpoints against that.

First, you can reduce the overhead from this by batching messages together. Since compression happens at the transmission level, more often than not the from and to keys will match from message to message. That means you rarely need to repeat those fields.

Second, this format allows you to verify it was sent by the public key given. It means that if you implement a system where certain stored values are ”owned” by a given node, it’s much easier to verify if the node requesting the change is allowed to.

Third, if we need to trade overhead for security, that can be a very worthwhile trade. True, it’s not necessary for everything, but that doesn’t mean there should be no balance between the two.

4.1.3 Message Payload

The message payload is an object encoded using the msgpack[3] standard. If the encryption bit is set, this section will be encrypted using the to public key.

[3] https://github.com/msgpack/msgpack/blob/master/spec.md

4.1.3.1 Limitations

In order to preserve the maximum compatibility, we impose additional restrictions on the types of objects that may be encoded. You may pack any of the following:

1. Nil
2. Booleans
3. Doubles (including NaN, Inf, and -Inf)
4. Integers from -(2^{63}) to (2^{64})-1
5. Strings smaller than length 2^{32}
6. Buffers smaller than length 2^{32}
7. Lists containing fewer than 2^{32} items
8. Maps containing fewer than 2^{32} associations, with string keys

This may be extended if the various msgpack libraries support serializing addiitonal types. At the time of writing this, timestamps have just entered the msgpack specification. They are largely unimplemented in the various msgpack libraries.

4.1.3.2 Why not JSON?

Partly because of licensing concerns, but mostly because in most languages, msgpack is faster. It's also significantly denser. Consider serializing the string `\x00\x00\x01\xff`, something you might do fairly often in this library.

JSON: `" \\ u 0 0 0 0 \\ u 0 0 0 0 \\ u 0 0 0 1 \\ u 0 0 f f "`

msgpack[4]: `\xc4 \x04 \x00 \x00 \x01 \xff`

That's 26 bytes to msgpack's 6.

4.1.4 Transmission Overview

Each transmission will start with a Transmission Header, and at least one pair of Message Header and Payload. Message Headers and Payloads *always* come in associated pairs, and they are *always* directly next to each other.

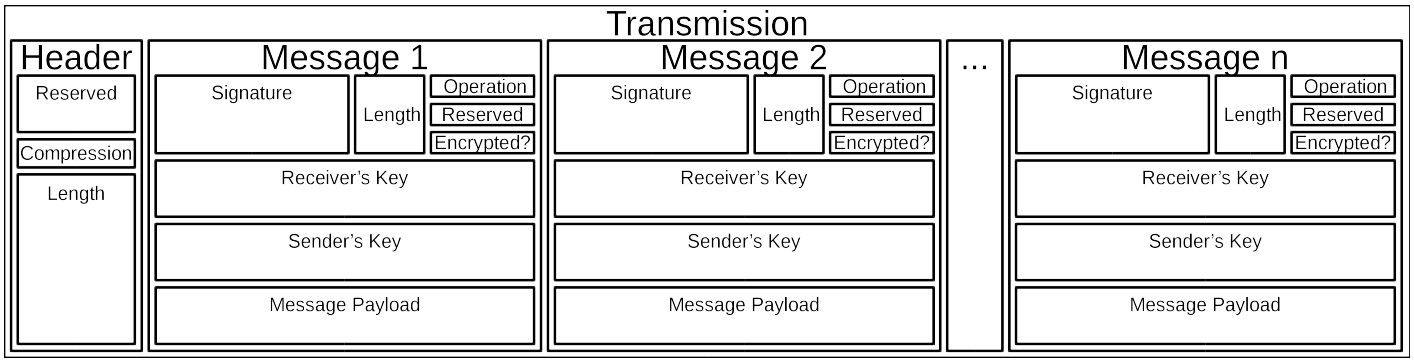

Figure 1: Graphical representation of a transmission

4.2 Serializing

Each step will be both explained, and written in a python-like pseudocode.

```
# Note that while you would ordinarily use classes for this, I will be using
# tuples for the sake of brevity

def make_tx(compression, *messages):  # type: (int, *bytes) -> bytes
    """Make a transmission from a collection of messages"""
    payload = b"".join(messages)  # type: bytes
    payload = compress(payload, compression)
    # packs a null byte, an unsigned byte, and a big-endian 32 bit
```

[4] https://github.com/msgpack/msgpack/blob/master/spec.md

```
    # unsigned int
    return struct.pack("!xBL", compression % 8, len(payload)) + payload


def make_msg(to,  # type: RSA_Key
             op,  # type: int
             payload,  # type: MsgPackable
             priv_key,  # type: RSA_Key
             encrypted=False  # type: bool
    ):  # type: (...) -> bytes
    """Constructs a serialized message"""
    msg_payload = msgpack.packb(payload)  # type: bytes
    msg_to = to.encode()  # type: bytes
    msg_from = priv_key.pub_key.encode()  # type: bytes
    msg_op = op % 16  # type: int
    if encrypted:
        msg_payload = to.encrypt(msg_payload)
    msg_len = len(msg_payload)  # type: int
    msg_no_sig = b"".join(
        # packs a big-endian 32 bit unsigned int, then an unsigned byte,
        # then a bool
        struct.pack("!LB?", msg_len, msg_op << 4, encrypted),
        msg_to,
        msg_from
    )
    msg_sig = priv_key.sign(msg_no_sig)
    return msg_sig + msg_no_sig
```

4.3 Deserializing

Each step will be both explained, and written in a python-like pseudocode.

```
def parse_tx(transmission):  # type: (bytes) -> Iterator(Tuple)
    """Splits one transmission into its message components"""
    # note: tx is short for transmission
    tx_opts = transmission[:2]  # type: bytes
    # Now we parse the length. Luckily the standard library can do that
    tx_len = struct.unpack("!L", transmission[2:6])[0]  # type: int
    tx_payload = transmission[6:]  # type: bytes
    tx_compression = tx_opts[1] % 8  # type: int

    # Here we will decompress only the first tx_len bytes
    tx_payload = decompress(tx_payload[:tx_len], tx_compression)
    to_parse = len(tx_payload)  # type: int
    parsed = 0  # type: int

    while parsed < to_parse:
        msg_header = tx_payload[parsed : parsed + 114 + 2*β]  # type: bytes
        parsed += 114 + 2 * β
        msg_sig = msg_header[:32]  # type: bytes
        # Now we parse the length. Luckily the standard library can do that
        msg_len = struct.unpack("!L", msg_header[32:36])[0]  # type: int
        msg_op = msg_header[36] >> 4  # type: int
        msg_encrypted = msg_header[37] & 1  # type: int
        msg_from = msg_header[38:76+β/8]  # type: bytes
        msg_to = msg_header[76+β/8:114+β/4]  # type: bytes
        msg_payload = tx_payload[parsed : parsed + msg_len]  # type: bytes
        parsed += msg_len
        # In production you would probably use a class, but for brevity's
```

```
        # sake, we'll yield a tuple here
        yield (msg_sig, msg_from, msg_to, msg_len, msg_encrypted, msg_payload)
```

After being split in this way, it will get sent on to the protocol parser to determine what to do with each message.

5 Object Overview

Please note that these are guidelines. Actual implementations can vary. In addition, parts of these guidelines will only work effectively if your language has either function pointers or first class functions.

5.1 Basic Structure

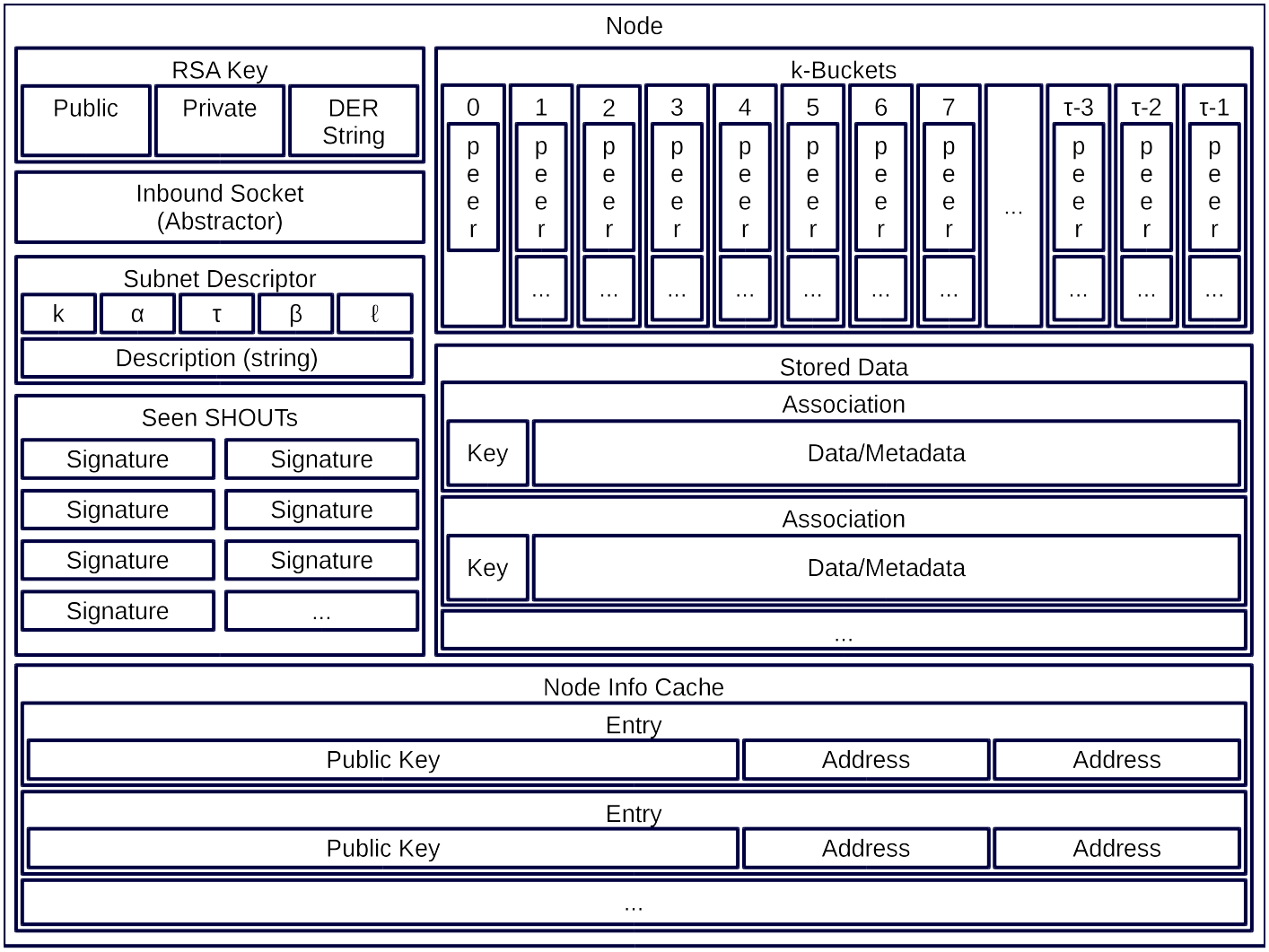

Figure 2: Object Diagram for a Network Node

5.1.1 RSA Key

5.1.2 Inbound Socket (Abstractor)

5.1.3 Subnet Descriptor

5.1.4 k-Buckets

5.1.4.1 Peer

5.1.5 Seen SHOUTs

5.1.6 Stored Data

5.1.6.1 Metadata

5.1.7 Node Info Cache

5.2 Protocol Parsing

5.3 Connection Management

6 RPCs

This section describes how your node should respond to incoming network messages

6.1 ACK [*]

This is the RPC that should be sent back to acknowledge a network message as successful, and provide return data if necessary.

Note that the arguments are contained in a list.

6.2 NACK [*]

This is the RPC that should be sent back to acknowledge a network message as failed, and provide return data if necessary.

Note that the arguments are contained in a list.

6.3 PING

Always respond with `ACK [PING]`. This will be utilized heavily in datagram protocols like UDP or µTP.

6.4 SET_CONNECTION_OPT [<option>, <setting>]

This will take two arguments. The first will be the option you wish to set, and the second is what you will set it to. Typically this will be something like enabling a compression method, or setting one as preferred.

Should either respond `ACK [SET_CONNECTION_OPT, <option>, <setting>]` or `NACK [SET_CONNECTION_OPT, <option>, <setting>]`, depending on if your node supports this setting.

Note that the arguments are contained in a list.

6.5 ANNOUNCE

This RPC is used to announce your presence to the network. It is relayed like SHOUT, and does not require an ACK.

6.6 CHANGE_KEY [<current key>, <new key>]

This RPC is used as a key change mechanism. Essentially, it allows you to change your public key every so often. This can be used to make it more difficult to impersonate a node. It is relayed like SHOUT, and does not require an ACK except from your direct peers.

6.7 SHOUT <message>

This indicates that a message should be forwarded to all peers if you have not previously seen it. ACKs are ill-advised here.

Assuming the above, and that ℓ is obeyed, we should be able to make some reasonable assumptions.

6.7.1 Defining Some Terms

```
n    number of nodes on the network
ℓ    the limit on outward connections
m    the number of messages per broadcast
t    sum(node.num_connections for node in nodes)
```

6.7.2 Special Case: Saturated Networks

This case is less efficient in most situations. Because each node can see all other nodes, we can say that it has (n - 1) connections. Each node will relay to all but one of its connections, except the original sender, who sends it to all. Therefore we can say:

```
t = (n - 1) × n
m = t - n + 1
= (n - 1) × n - n + 1
= n^2 - 2n + 1
= (n - 1)^2
= 0(n^2)
```

6.7.3 Special Case: Limited Networks

A limited network is where each node has ℓ outward connections. This is the limit set in software, so a node will not initiate more than ℓ connections on its own. Because connections must have another end, we can conclude that the average number of inward connections per node is also ℓ. Therefore:

```
t = 2ℓ × n
m = t - n + 1
= 2ℓ × n - n + 1
= (2ℓ - 1) × n + 1
= 0(n)
```

6.7.4 Crossover Point

You should be able to show where these two domains meet by finding the point where m is equal.

```
(n - 1)^2 = (2ℓ - 1) × n + 1
n^2 - 2n + 1 = (2ℓ - 1) × n + 1
n^2 - 2n = (2ℓ - 1) × n
n - 2 = 2ℓ - 1
n = 2ℓ + 1
```

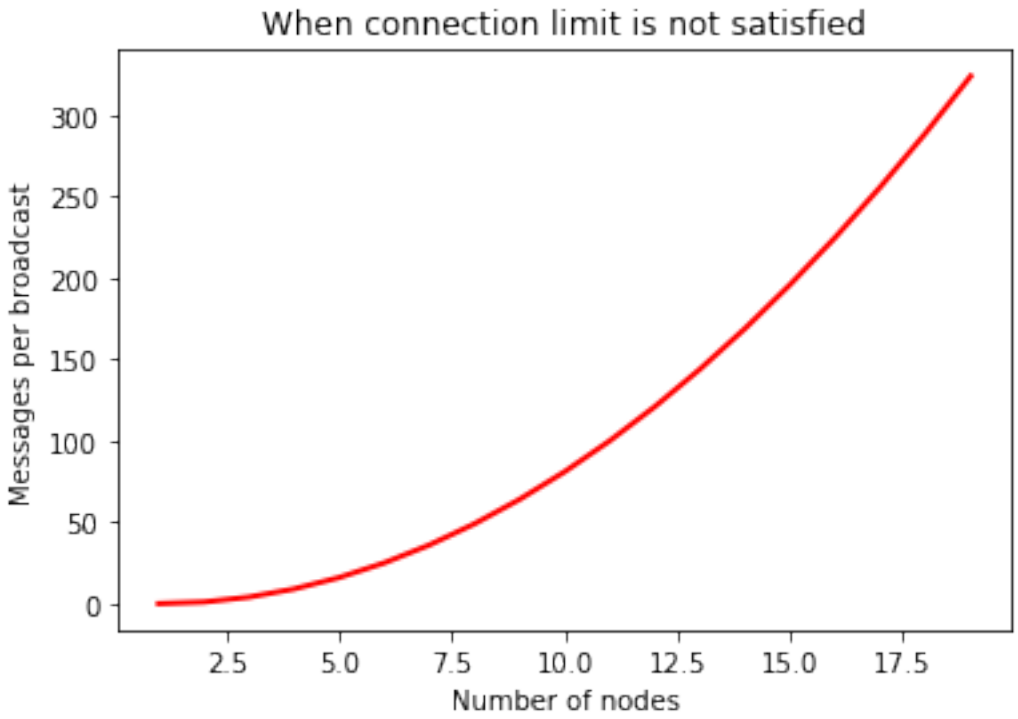

Figure 3: Data sent to nodes on a network for a single broadcast in saturated networks

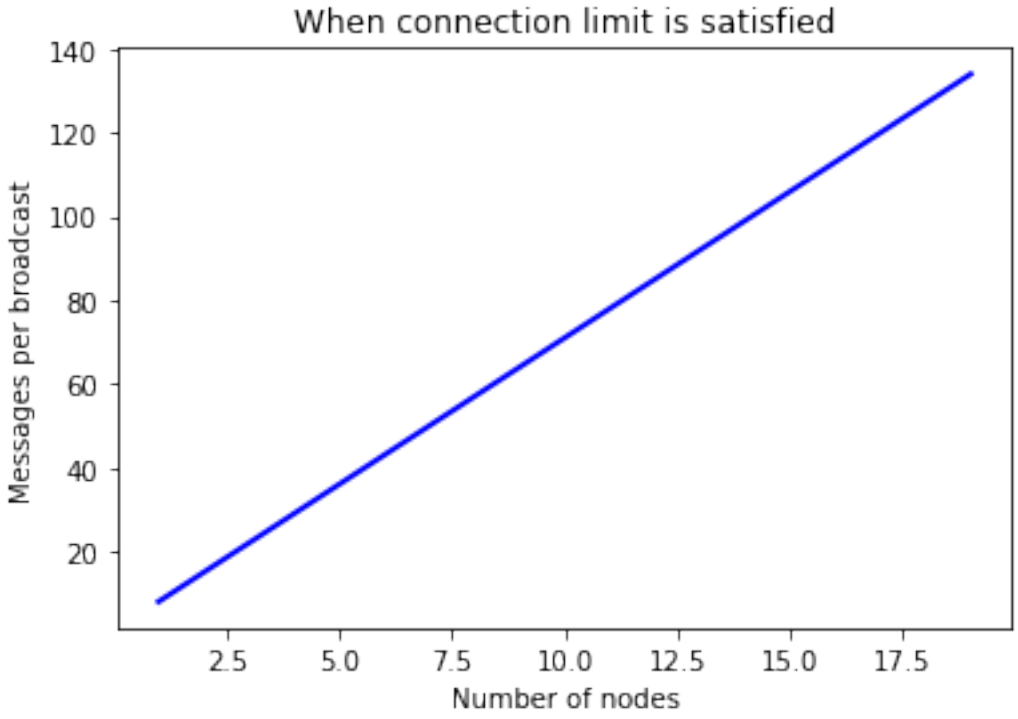

Figure 4: Data sent to nodes on a network for a single broadcast in limited networks

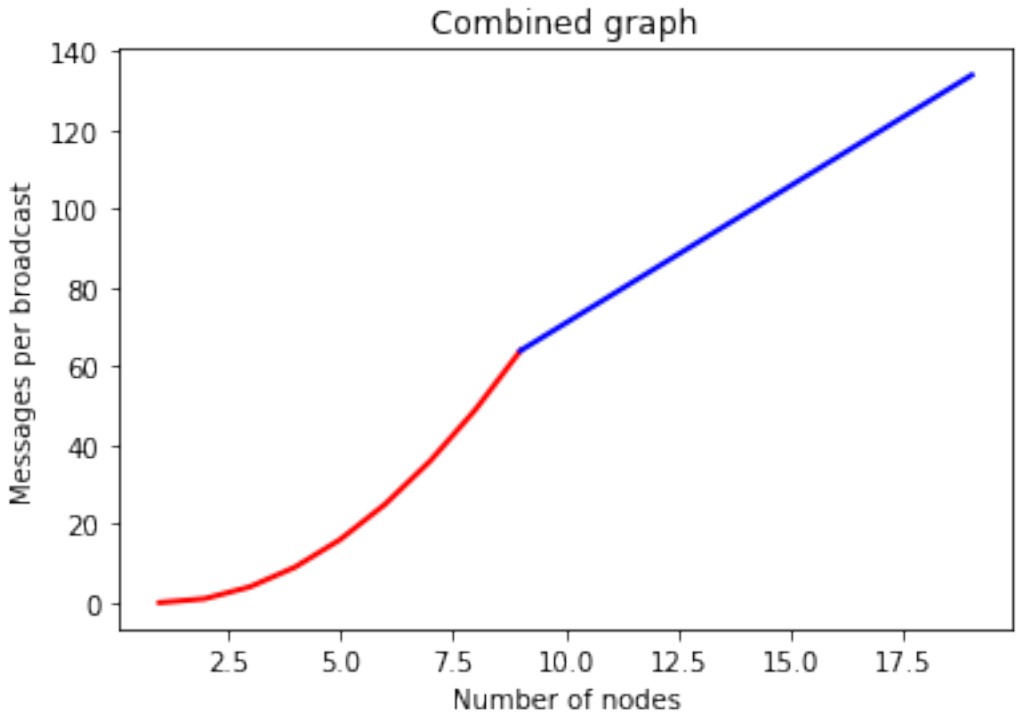

Figure 5: Data sent to nodes on a network for a single broadcast

6.7.5 Lag Analysis

I managed to find the worst possible network topology for lag that this library will generate. It looks like figures 6 and 7.

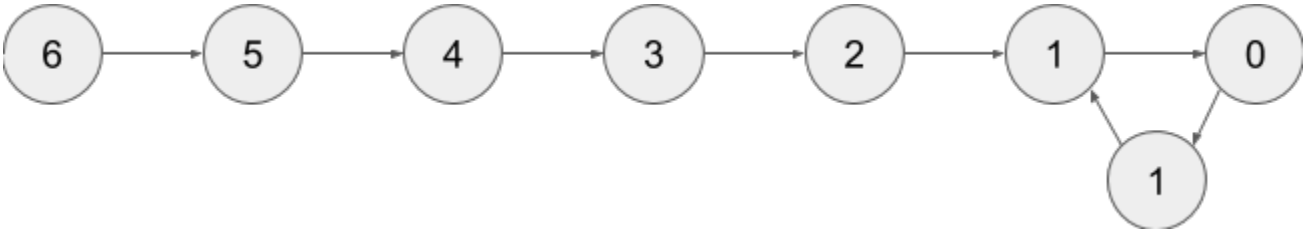

Figure 6: Delay in hops for a worst-case network with ℓ=1

The lag it experiences is described by the following formula (assuming similar bandwidth and latency):

`lag = ceil(max((n-2) ÷ ℓ, 1)) for all networks where n > 2ℓ + 1`

6.7.6 Conclusion

From this, we can gather the following:

1. For all networks where n < 2ℓ + 1, m is Θ(nˆ2)
2. For all networks where n >= 2ℓ + 1, m is Θ(n)
3. All networks are O(n)
4. Lag follows ceil(max((n-2) ÷ ℓ, 1))

6.7.7 Comparison to Centralized Architecture

When comparing to a simplified server model, it becomes clear that there is a fixed, linearly scaling cost for migrating to this peer-to-peer architecture.

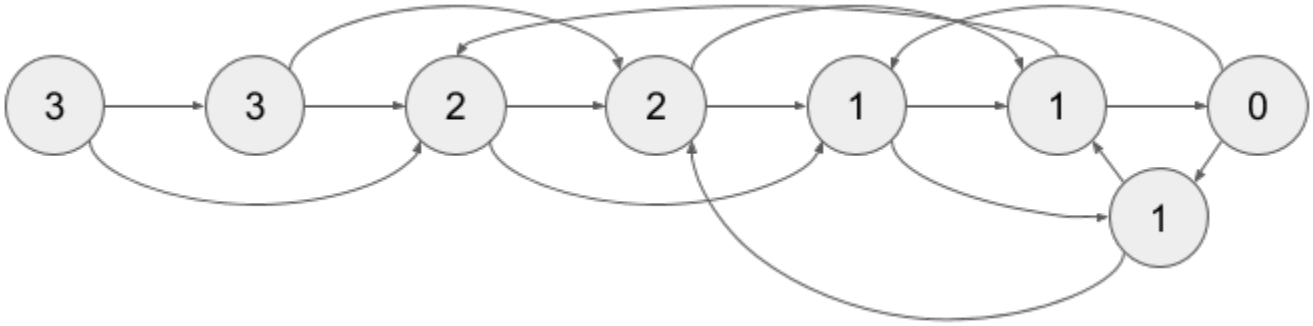

Figure 7: Delay in hops for a worst-case network with ℓ=2

The model we'll compare against has the following characteristics:

1. When it receives a message, it echoes it to each other client
2. It has ℓ threads writing data out
3. Each client has similar lag and bandwidth

Such a network should follow the formula:

`lag = ceil((n-1) ÷ ℓ) + 1`

This means that, for any network comparison of equal ℓ and n, you have the following change in costs:

1. Worst case lag is *at worst* the same as it was before (ratio ≤ 1)
2. *Total* bandwidth used is increased by a factor of 2ℓ - 1 + (1 ÷ n)

Therefore, we can conclude that this broadcast design satisfies the given requirements for an efficient protocol.

6.8 SPEAK <message>

This indicates that a message may be forwarded to all peers *at your discretion*, if you have not previously seen it. By default a node should *not* forward it, but there are some situations where it might be desirable.

ACKs are not necessary except on UDP-like transports, since the nodes receiving this message are directly connected. If it is difficult to implement this conditional, send the ACK by default.

6.9 WHISPER <message>

This indicates that a message is intended for a specific destination. The message may or not be encrypted. That should be handled on the message parser level.

Acknowledge these messages in the format `ACK [WHISPER, <message signature>]`.

6.9.1 If Directly Connected

Send the message directly. Encrypt if using an insecure transport method. Otherwise encryption is optional.

6.9.2 If Not Directly Connected

Otherwise things can be ambiguous. Both of these methods should be supported, but the decision on which to take should be made locally.

6.9.2.1 Iterative

This strategy should be preferred if your k-buckets are not yet filled. Essentially you should issue `FIND_NODE` RPCs until you've received the info for the node you are looking for. When this has happened, send directly. Under this scheme, encryption follows the same rules as if you are directly connected, because you will be.

6.9.2.2 Recursive

This strategy should be preferred if your k-buckets *are* filled. To do this, you issue a `WHISPER` RPC to the closest node you have. They will then follow this same decision tree. In this scheme encryption is *mandatory*.

6.10 FIND_NODE <extended address>

This is mostly defined by the Kademlia[5] spec. Essentially, they send you an address, and you reply with the `k` closest nodes you're aware of to that address, where distance is given by `XOR(<extended address>, addr) % 2**τ`. If you don't know of `k` nodes, send back as many as are known. Format like `ACK [FIND_NODE, <node 0 info>, <node 1 info>, ...]`.

6.11 FIND_VALUE [<truncated address>, <key>]

While the address can be computed directly from the key, both are included to save computation time.

Note that the arguments are contained in a list.

6.11.1 If Value Unknown

Respond as if it was a `FIND_NODE` RPC.

6.11.2 If Value Known

Respond in the format `ACK [FIND_VALUE, <key>, <value>, <metadata>]`. Metadata is defined in the Object Overview section.

6.12 STORE [<truncated address>, <key>, <value>]

While the address can be computed directly from the key, both are included to save computation time. It should `ACK` in a similar format to `FIND_VALUE`.

Note that the arguments are contained in a list.

6.13 CUSTOM <message>

This is the opcode reserved for building on top of this protocol. Part of the public API is a way to hook into the protocol parser. This opcode indicates that a message is meant for this part of the API, rather than a part of the protocol defined above.

7 Public API

[5] https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf