Gabe Appleton

Contents

1 The p2p.today Project **2**
- 1.1 Purpose . . . 2
- 1.2 Implementation Details . . . 2
  - 1.2.1 Why not JSON? . . . 3
  - 1.2.2 What transport layers can it use? . . . 3
- 1.3 So how is everything structured? . . . 3
  - 1.3.1 Serializing Container . . . 3
  - 1.3.2 Event Loop . . . 5
  - 1.3.3 Protocol Queue . . . 5
  - 1.3.4 Subnet Descriptors . . . 6
- 1.4 The Bootstrap Function . . . 6

2 Mesh Networking **6**
- 2.1 Structural Changes . . . 6
- 2.2 Routing . . . 6
  - 2.2.1 Direct Messages . . . 6
  - 2.2.2 Broadcasts . . . 8
- 2.3 Initializing . . . 8
- 2.4 Linearly Scalable Broadcasts . . . 8
  - 2.4.1 Defining Some Terms . . . 8
  - 2.4.2 Special Case: Saturated Networks . . . 9
  - 2.4.3 Special Case: Limited Networks . . . 9
  - 2.4.4 Crossover Point . . . 10
  - 2.4.5 Lag Analysis . . . 12
  - 2.4.6 Conclusion . . . 12
  - 2.4.7 Comparison to Centralized Architecture . . . 12

3 Synchronized Dictionaries **12**
- 3.1 Use Case . . . 12
- 3.2 Structural Changes . . . 12
- 3.3 Initializing . . . 13
- 3.4 How it works . . . 13
  - 3.4.1 Setting Data . . . 13
  - 3.4.2 Leasing . . . 13
  - 3.4.3 Applying a Delta . . . 14

4 Distributed Hash Table **14**
- 4.1 Use Case . . . 14
- 4.2 Structural Changes . . . 14
- 4.3 Initializing . . . 14
- 4.4 How it works . . . 15
  - 4.4.1 The Chord Algorithm . . . 15
  - 4.4.2 Setting Data (Redux) . . . 15
  - 4.4.3 Applying a Delta (Redux) . . . 15

5 Dependencies **16**
- 5.1 Python . . . 16
- 5.2 Testing And Documentation . . . 16
- 5.3 Node.js . . . 16
- 5.4 Browser Javascript . . . 17

6 Rubric and Grading Scale **17**
- 6.1 Rubric . . . 17

6.2 Grading Scale . . . 18

7 Additional Information **18**
7.1 To Install . . . 18
7.2 Source Code . . . 18
7.3 Documentation . . . 18

List of Figures

1 Object Hierarchy Template for the Network Socket Abstractor . . . 4
2 Object Hierarchy for the Mesh Network Socket Abstractor . . . 7
3 Data sent to nodes on a network for a single broadcast in saturated networks . . . 9
4 Data sent to nodes on a network for a single broadcast in limited networks . . . 10
5 Data sent to nodes on a network for a single broadcast . . . 11
6 Delay in hops for a worst-case network with ℓ=1 . . . 11
7 Delay in hops for a worst-case network with ℓ=2 . . . 11

1 The p2p.today Project

1.1 Purpose

Peer-to-peer networking is hard. Especially if you want to run between multiple languages. When it comes down to it, there are very few libraries that allow you to easily communicate serialized values to a different language. Most of the time you have to either write it yourself, or wrap it in JSON.

This set of libraries aims to provide a flexible API which manages connections, deserializes messages, and allows easy automation of actions, while keeping everything as portable as possible.

Currently it has full support in:

- Python
- Node.js v5+ JavaScript
- Browser JavaScript (no incoming connections)

And serialization support in:

- C
- C++

The previous version of this library also had serialization support for:

- Golang
- Java
- Smalltalk

However, due to my lack of experience in those languages, they were temporarily abandoned.

The underlying dependencies are implemented in ~30 languages right now, including C#, Swift, Ruby, Clojure, Haskell, Lua, and Scheme.

1.2 Implementation Details

This library uses msgpack[1] for serialization.

[1] https://msgpack.org

1.2.1 Why not JSON?

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: `\xc4 \x04 \x00 \x00 \x01 \xff`

That's 26 bytes to msgpack's 6.

Because we do this there are some limitations. It means this library can serialize:

1. Nil
2. Booleans
3. Doubles (including NaN, Inf, and -Inf)
4. integers from -(2^{63}) to (2^{64})-1
5. strings up to length (2^{32})-1
6. buffers up to length (2^{32})-1
7. Arrays containing up to (2^{32})-1 items
8. Maps containing up to (2^{32})-1 associations

1.2.2 What transport layers can it use?

Theoretically it can use anything which provides socket-like interfaces. Currently it supports TCP, SSL/TLS, and WebSockets (JavaScript only).

Due to the how greatly Python WebSocket libraries differ from Python sockets, adding support for that would require largely rewriting the library's event loop. It's planned, but not anytime in the near future.

1.3 So how is everything structured?

All network implementations are based on structure shown in Figure 1.

1.3.1 Serializing Container

This object serves three functions. First, it acts as a dumb data store. Second, it knows how to serialize a message. And third, it knows how to compress a message, given a sequence of possible method flags.

1.3.1.1 Headerless Serialization

The barest possible serialization is given by the following JavaScript:

```
let to_serialize = [msg.type, msg.sender, msg.time, ...msg.payload];
let bare_serialized = msgpack.encode(to_serialize);
```

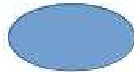
1.3.1.2 Checksum

This is a calculated property. It should be processed by taking the SHA256 hash of the above serialization.

Legend:

Object

*Excepting JavaScript

Presented
To User 

Socket
Abstractor

Message
Queue

Protocol
Queue

Event
Loop*

Subnet
Descript.

Routing
Table

Nodes
Awaiting
Handshake

Message

Protocol
Function

Event
Callbacks

Association

Serializing
Container

Peer ID

Connection

Checksum

Type
Flag

Time

Sender ID

Payload
Array

Socket

Time Last
Received

Message
Last Sent

Supported
Compress.

Peer
Address

Peer ID

Figure 1: Object Hierarchy Template for the Network Socket Abstractor

1.3.1.3 Serialization With Headers

Full serialization requires five steps.

1. Calculate the above properties
2. Concatenate them with the digest first
3. If negotiated, compress this buffer
4. Encode the buffer's length in 4 big endian bytes
5. Prepend that to the buffer

1.3.2 Event Loop

There are two events to consider for this library.

1.3.2.1 Incoming Connection

On incoming connection, the socket abstractor should immediately send a handshake message, and add this connection to the collection of nodes awaiting handshake.

1.3.2.2 Node Sends Data

When this happens, the socket abstractor should go through the process detailed in the next section. One of these protocol functions should know how to handle a handshake, and in doing so add it to the routing table.

1.3.3 Protocol Queue

Essentially, there are three interfaces for dealing with messages:

1. Immediate access as a protocol parser
2. Filtered access as an Event callback
3. Occasional calls to `connection.recv()`

1.3.3.1 Protocol Functions

Protocol functions are where all the major processing happens. While a user *can* add one themselves, it is generally discouraged.

This is where all relaying and local storage is enabled. It is also where the least restrictions are placed. The library only ensures that the function can be called without erroring (e.g. wrong number of arguments in Python).

Every protocol is used by successively calling these functions until:

1. It throws an error, in which case call the relevant method
2. It returns true, in which case you should stop protocol parsing
3. All methods have been called, in which case you should add it to the message queue, and trigger the `'message'` event

1.3.3.2 Event Callbacks

This is a set of callbacks registered through an EventEmitter interface. Essentially, a user says `connection.on(some_message, some_callback)`, and after filtering messages, this library will let them know about it.

The restriction put on this set of events is that they must call `connection.recv()` in order to obtain the actual message. This is meant to stop memory leaks from occurring.

1.3.3.3 `connection.recv([max_return = 1])`

This would be analogous to calling `read()` on a socket. This function has three defined behaviors.

1. If you supply the argument max_return, return a list of `Message` objects at most that long
2. If you do not supply the argument, and there are `Messages`, return the oldest
3. If you do not supply the argument, and there are no `Messages`, return `None`

The use case here is for serial processes or scripts. It is not recommended for real applications.

1.3.4 Subnet Descriptors

This object describes what set of nodes you want connections to, and it's used to filter out undesired connections. Essentially, you feed it some descriptive string and the transport method you'd like to use. You can then ask it to calculate the ID of this subnet, which is returned as a base58 encoded SHA256 digest.

1.4 The Bootstrap Function

This is a big part of what makes the library useful. Essentially, the bootstrap function queries a network to see if any of your desired peers are online. It does the following:

1. Use the given constructor and arguments to make a socket abstractor
2. Start an asynchronous process
3. Return the socket
4. In the background, connect to the bootstrap network
5. Query it for the given subnet descriptor's ID
6. For each address returned (shuffled, up to some limit), try connecting the new socket to it

You'll see an example of it in the next section.

2 Mesh Networking

Let's say you have a soccer team. Each robot needs only to signal its moves to its peers. This is a perfect use case for the mesh network.

2.1 Structural Changes

Other than registering some protocol functions, there is only one addition to a mesh socket: a collection of recently seen messages. See Figure 2 for a modified object diagram.

2.2 Routing

This network architecture provides two major routing methods.

2.2.1 Direct Messages

If you are directly connected to a peer, you can send them a "whisper". If you are not directly connected, you can send a request for the address of some peer ID. This request is not guaranteed to be answered, as they may have disconnected.

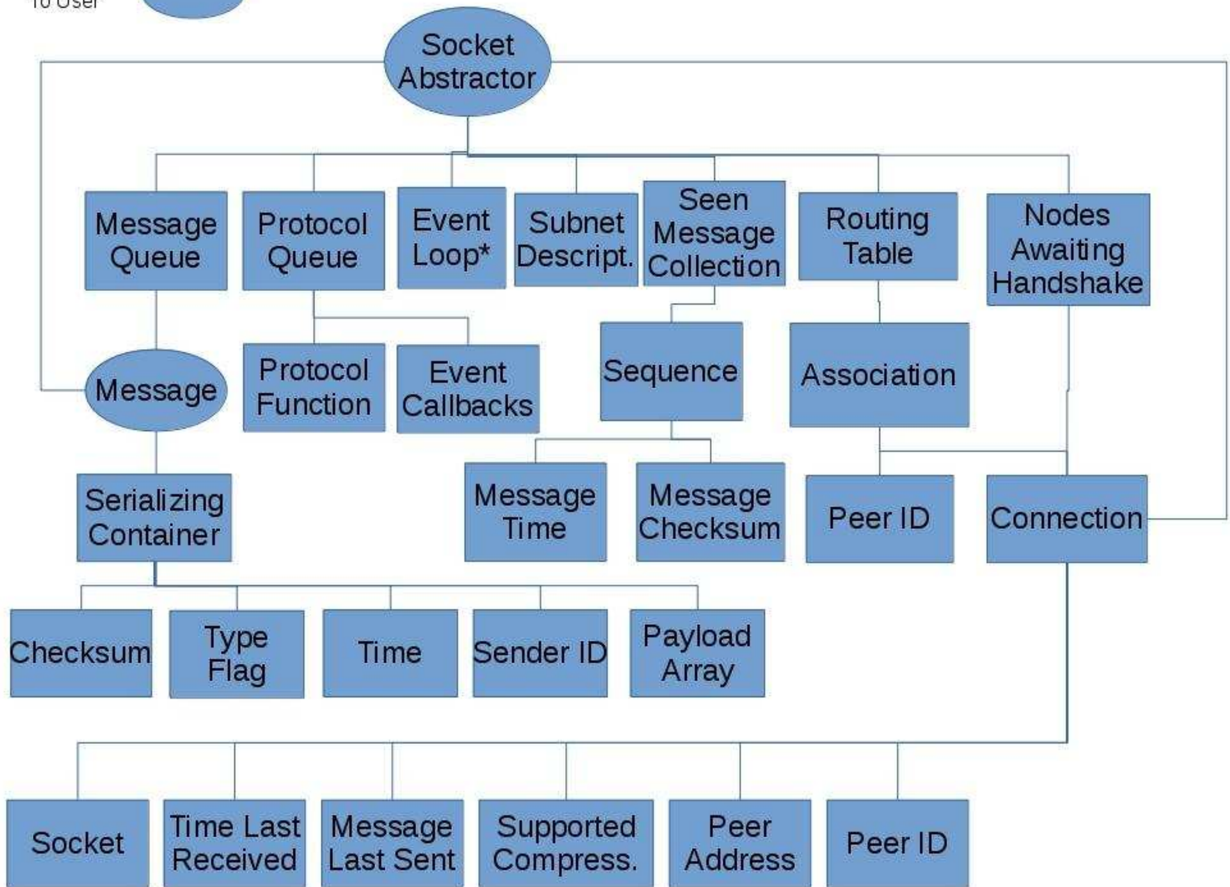


Figure 2: Object Hierarchy for the Mesh Network Socket Abstractor

2.2.2 Broadcasts

The other choice is to broadcast to *all* connected nodes. This is done through a flooding algorithm, but as I'll show in Section 2.4, it is modified to be linearly scaling.

2.3 Initializing

You may initialize using one of the methods shown below. An example is shown for both the bootstrap function and actual construction.

```python
# Python
import py2p
# Object construction
conn = py2p.mesh.MeshSocket(
    '0.0.0.0',
    44444,
    py2p.base.Protocol('example', 'SSL')
)
# Or with bootstrap()
conn = py2p.bootstrap(
    py2p.mesh.MeshSocket,
    py2p.base.Protocol('soccer team', 'Plaintext'),
    '0.0.0.0',
    44565
)
```

```javascript
// Javascript
let js2p = require('js2p');
// Object construction
let conn = new js2p.mesh.MeshSocket(
    '0.0.0.0',
    44444,
    new js2p.base.Protocol('example', 'SSL')
)
// Or with bootstrap()
conn = js2p.bootstrap(
    js2p.mesh.MeshSocket,
    new js2p.base.Protocol('soccer team', 'Plaintext'),
    '0.0.0.0',
    44565
);
```

2.4 Linearly Scalable Broadcasts

There are two basic states in this type of network, saturated and limited.

Limited networks are where the software *will not* allow it to make any additional outbound connections. Saturated networks are those where each node can see every other node.

If the limit on outbound connections is ℓ, then we can figure out how many messages the network will send on a given broadcast.

2.4.1 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)
```

2.4.2 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:

$$t = (n-1) \times n$$
$$m = t - n + 1$$
$$= (n-1) \times n - n + 1$$
$$= n^2 - 2n + 1$$
$$= (n-1)^2$$
$$= \Theta(n^2)$$

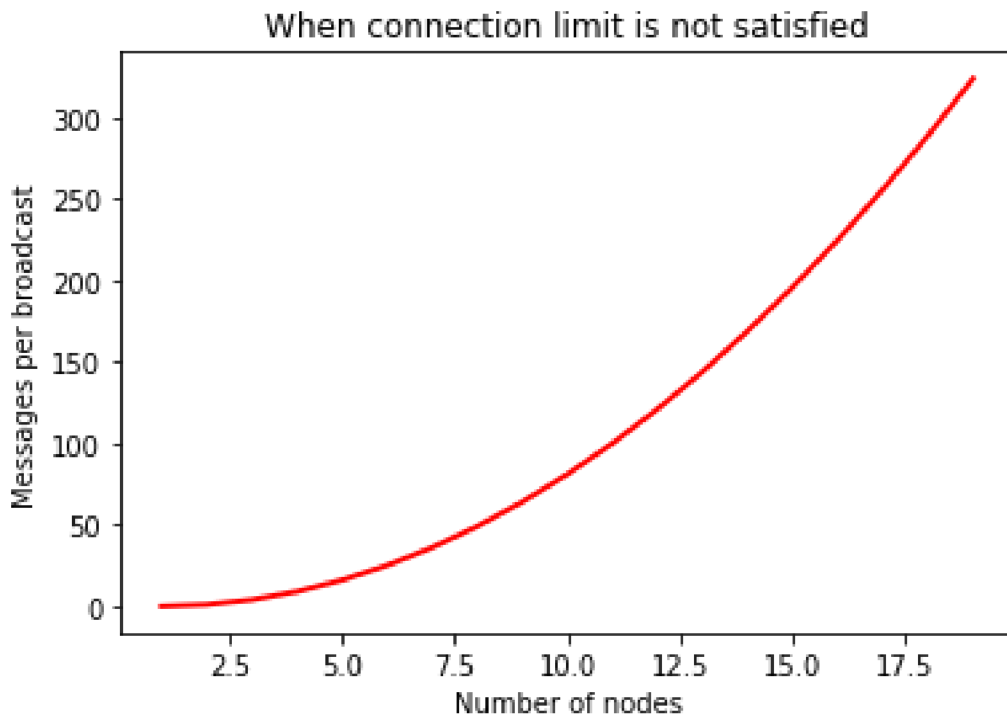


Figure 3: Data sent to nodes on a network for a single broadcast in saturated networks

2.4.3 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 number of inward connections per node is also ℓ. Therefore:

$$t = 2\ell \times n$$
$$m = t - n + 1$$
$$= 2\ell \times n - n + 1$$
$$= (2\ell - 1) \times n + 1$$
$$= \Theta(n)$$

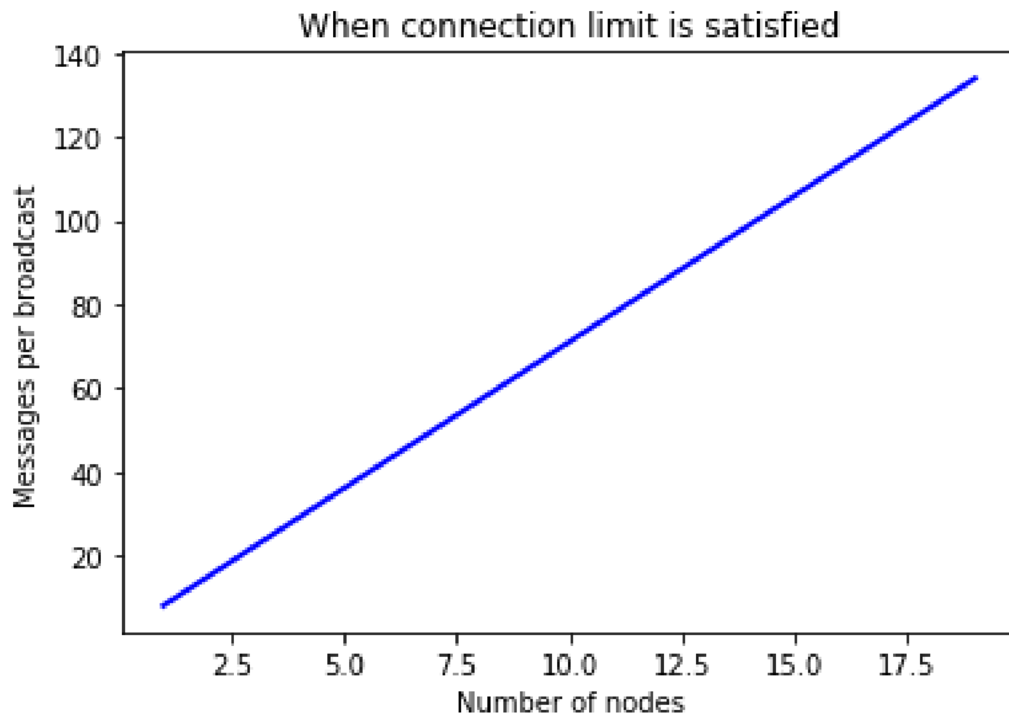


Figure 4: Data sent to nodes on a network for a single broadcast in limited networks

2.4.4 Crossover Point

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

$$(n - 1)^2 = (2\ell - 1) \times n + 1$$
$$n^2 - 2n + 1 = (2\ell - 1) \times n + 1$$
$$n^2 - 2n = (2\ell - 1) \times n$$
$$n - 2 = 2\ell - 1$$
$$n = 2\ell + 1$$

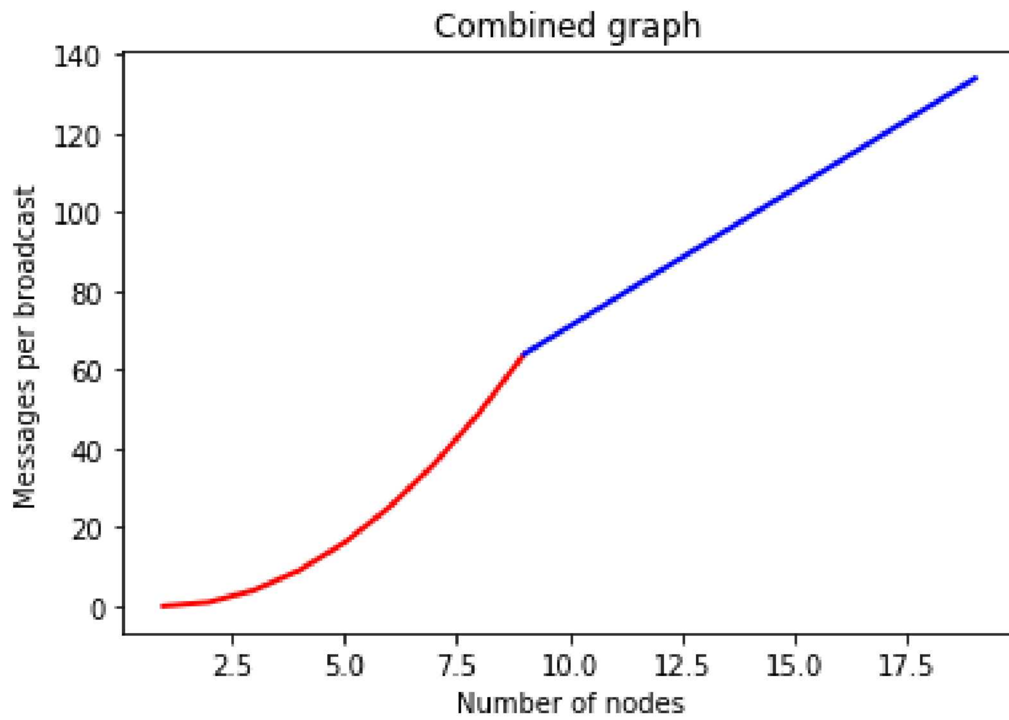


Figure 5: Data sent to nodes on a network for a single broadcast

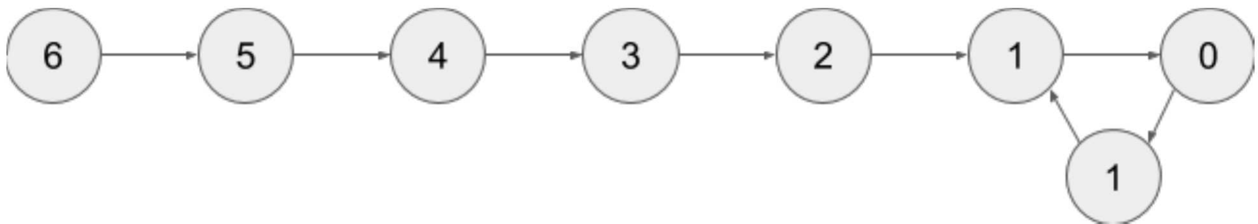


Figure 6: Delay in hops for a worst-case network with ℓ=1

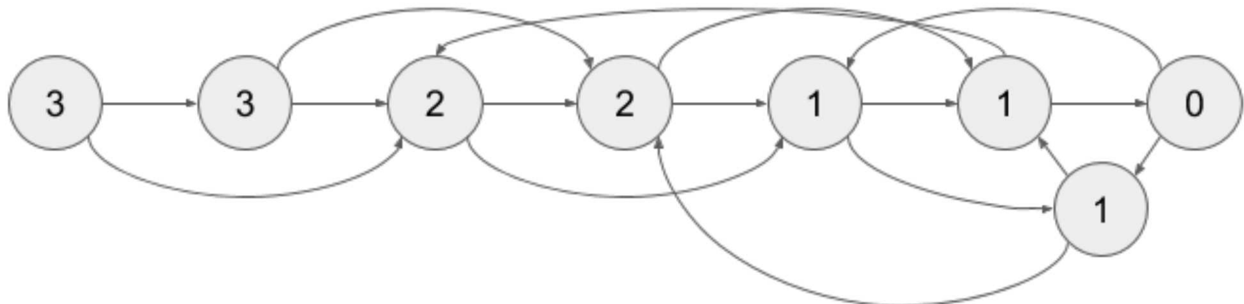


Figure 7: Delay in hops for a worst-case network with ℓ=2

2.4.5 Lag Analysis

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

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

$$lag\ factor = \lceil max((n-2) \div \ell, 1) \rceil \text{ for all networks where } n > 2\ell + 1$$

2.4.6 Conclusion

From this, we can gather the following:

1. For all networks where n < 2ℓ + 1, m is Θ(n^2)
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))

2.4.7 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.

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:

$$lag\ factor = \lceil (n-1) \div \ell \rceil + 1$$

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

3 Synchronized Dictionaries

3.1 Use Case

You are a simple machine, and you don't need to keep track of many associations. You are a combat Roomba. Your mission in life is to find the table leg, and destroy it, but you don't have the CPU cycles to spare on that hashing non-sense.

Additionally, you require *fast* database access. Network querying? Way too long. You need to know the value of an item, and you need to know it synchronously.

3.2 Structural Changes

Very little is changed from the mesh socket. The vast majority is in adding extra protocol functions. All the rest is in two new dictionaries: `connection.data` and `connection.metadata`.

There are also several accessor methods added, including a batch update that iterates over a dictionary.

3.3 Initializing

```
# Python
import py2p
# Object construction
conn = py2p.sync.SyncSocket(
    '0.0.0.0',
    44444,
    py2p.base.Protocol('example', 'SSL')
)
# Or with bootstrap()
conn = py2p.bootstrap(
    py2p.sync.SyncSocket,
    py2p.base.Protocol('soccer team', 'Plaintext'),
    '0.0.0.0',
    44565
)
```

```
// Javascript
let js2p = require('js2p');
// Object construction
let conn = new js2p.sync.SyncSocket(
    '0.0.0.0',
    44444,
    new js2p.base.Protocol('example', 'SSL')
)
// Or with bootstrap()
conn = js2p.bootstrap(
    js2p.sync.SyncSocket,
    new js2p.base.Protocol('soccer team', 'Plaintext'),
    '0.0.0.0',
    44565
);
```

3.4 How it works

3.4.1 Setting Data

The simpler message send is the `store` command. Essentially, it broadcasts the key and the new data. Metadata is extracted from the message itself to determine owner and time of change.

If you are initially syncing to the network, there are two other items added: the ID of the last changer, and the timestamp of this change.

3.4.2 Leasing

By default this class implements a key leasing system. This system has two purposes:

1. Settle race condition ties
2. Keep people from changing a key that "belongs" to you (can be disabled)

To determine this, it goes through a series of checks:

1. If the key is not set, you have permission
2. If you are the last to change the key, you have permission

3. If you are applying a delta, and ownership is disabled, you have permission
4. If the last change was more than an hour ago, you have permission
5. If you submit a change within a second of someone else, the winning ID of a string compare has permission
6. If ownership is disabled, and the last change was in the past, you have permission
7. Otherwise, you do not have permission

3.4.3 Applying a Delta

This message has some strings attached. It *only* works if:

1. The value you are changing is a mapping, or
2. The value you are changing is not set

In all other cases, or if the lease check fails, it should throw an error.

Essentially this message allows you to update a nested dictionary. So it would translate to:

```
d = connection.get('test', {})  # returns a value, or an empty dictionary
d.update(some_dictionary)
connection.set('test', d)
```

This is implemented as a data saving technique. Instead of transmitting the entire dictionary, which may be quite large, you can instead send only the portion you are changing.

4 Distributed Hash Table

4.1 Use Case

This distributed database is good for cases where you need to store large databases. It will typically be better if you need infrequent access to this database, since it will move at the speed of internet lag.

4.2 Structural Changes

The chord abstractor adds three items:

1. A boolean indicating whether you "leech" from the network
2. Your ID converted to a big integer
3. A nested dictionary of the following structure: `{'sha1': {some_hash: some_value}, 'sha256': {some_hash: some_value}, ...}`

4.3 Initializing

```
# Python
import py2p
# Object construction
conn = py2p.chord.ChordSocket(
    '0.0.0.0',
    44444,
    py2p.base.Protocol('example', 'SSL')
)
# Or with bootstrap()
conn = py2p.bootstrap(
    py2p.chord.ChordSocket,
```

```
    py2p.base.Protocol('soccer team', 'Plaintext'),
    '0.0.0.0',
    44565
)

// Javascript
let js2p = require('js2p');
// Object construction
let conn = new js2p.chord.ChordSocket(
    '0.0.0.0',
    44444,
    new js2p.base.Protocol('example', 'SSL')
)
// Or with bootstrap()
conn = js2p.bootstrap(
    js2p.chord.ChordSocket,
    new js2p.base.Protocol('soccer team', 'Plaintext'),
    '0.0.0.0',
    44565
);
```

4.4 How it works

4.4.1 The Chord Algorithm

The Chord algorithm was created by a group at MIT[2]. Essentially, it dictates the following rules:

1. Distance between two nodes is dictated by `(b - a) % max_id`
2. Ideally, you should have connections to each node `2^n` away from you
3. To get data, ask the nearest known connection to that

In the original chord algorithm they suggest doing this iteratively, however the current implementation in these libraries does it recursively (think iterative vs recursive DNS). In addition, there are *five* of these tables, each using a different hash. This ensures that if a node suddenly exits everything should remain intact.

Following these rules should get you `O(log(n))` lookup speeds.

4.4.2 Setting Data (Redux)

This isn't hugely different from the sync table described above. The big difference is that instead of a key, you give a method and a hash (encoded in base58). This message then gets relayed until it gets to the responsible node.

4.4.3 Applying a Delta (Redux)

Again, this isn't hugely different from the sync table. The real difference is that, because there's less metadata being kept about these changes, you could end up in a situation where the key gets corrupted from the different internal tables disagreeing with each other.

The upside is that the internal use case is not impacted by this, as this condition doesn't happen if you update keys which are either unique or have the same value.

[2] https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

5 Dependencies

Bolded names mean that I had contributions to that library.

5.1 Python

Required:

- **custom_inherit** (Documentation inheritance)
- **async_promises** (My fork of **promise**, a port of JavaScript's A+ Promises)
- u-msgpack-python (Serialization)
- **pyee** (EventEmitter API)
- typing (Easier support for other libraries)
- **base58** (Base58 encoding)
- click (Running the shell interface)

Optional:

- cryptography (TLS/SSL support)
- python-snappy (Faster compression)

5.2 Testing And Documentation

- Travis CI (OSX testing)
- Shippable (Linux testing)
- Appveyor (Windows testing)
- Codecov.io (Measuring code coverage)
- **Netlify** (Hosting documentation)
- sphinx (Building documentation)
- sphinxcontrib-napoleon
- sphinx_rtd_theme
- codecov (Reporting code coverage)
- pytest (Running python tests)
- pytest-coverage
- pytest-benchmark
- pytest-ordering
- mocha (Running JavaScript tests)
- istanbul (Getting JavaScript coverage)

5.3 Node.js

Required:

- big-integer (Large integers for hashes)
- equal (Object equality)
- ip (Fetches your LAN IP)
- jssha (Dense hash library)
- msgpack-lite (serialization)
- vorpal (Running the REPL interface)
- yargs (Running the shell interface)

Optional:

- node-forge (SSL/TLS support)

- nodejs-websocket (Websocket support)
- snappy (Faster compression)
- zlibjs (Zlib compression)
- babel (Transpiling for older environments)
- babel-runtime
- babel-plugin-transform-object-rest-spread
- babel-plugin-transform-runtime
- babel-preset-es2015
- babel-preset-latest

5.4 Browser Javascript

Required:

- browserify (Transpiling for browsers)
- big-integer
- equal
- jssha
- msgpack-lite

Optional:

- zlibjs
- babel
- babel-runtime
- babel-plugin-transform-object-rest-spread
- babel-plugin-transform-runtime
- babel-preset-es2015
- babel-preset-latest

6 Rubric and Grading Scale

6.1 Rubric

Feature	**Points**	**Earned**
Python 3 support	20	20
Python 2 support	10	10
Node.js 4+ support	20	15
JavaScript buildable for browsers	10	10
Code partitioning for browser	10	10
Broadcast Network Type	20	20
Synchronized Dictionary Type	10	10
Distributed Dictionary Type	30	30
Opportunistic compression	10	10
Support for more primitives than bytes/Buffer	15	15
Bootstrapping function (find network by ID, not address)	10	6
Simple example application	15	15
PEP8 compliance in Python code (https://www.python.org/dev/peps/pep-0008)	10	10
ESLint compliance in Javascript code (http://eslint.org/docs/rules/)	10	10
Total	**200**	**191**

6.2 Grading Scale

```
A:    165 points          B:    145 points          C:    125 points
D:    105 points          F:    <85 points
```

7 Additional Information

7.1 To Install

Python:

```
pip install py2p  # default
pip install py2p["SSL"]  # with SSL support
pip install py2p["snappy"]  # with snappy compression
pip install py2p["SSL","snappy"]  # with both
```

Node.js:

```
npm install js2p
```

7.2 Source Code

http://git.p2p.today

7.3 Documentation

https://dev-docs.p2p.today