P2PD

Release 0.1

Matthew Roberts

Feb 16, 2023

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 The problem with P2P networking

In peer-to-peer networking one of the most basic problems is trying to get two computers to connect to each other. The problem is that regular people's computers aren't designed to be used as servers. They are most often behind a router that employs NAT to control which connections can reach devices in the internal network. The router must be specifically configured to forward connections to the right machines – a process called 'port forwarding.'

Sometimes port forwarding can be done automatically. Protocols like UPnP and NATPMP allow programs to request a router to setup rules on its behalf. But in practice the availability of these features is not guaranteed. If you've ever tried to host a server in an online game before you will know this first hand. The software will have tried to use UPnP or NATPMP to port forward your server so that others can reach it. The trouble is that these features can be seen as a security risk and disabled by default.

Peer-to-peer direct connectivity is hard to do because of the number of possible network configurations, software platforms, and obstactles that must be bypassed to do it reliabily. It's for this reason that the most successful P2P systems don't actually solve this problem. Take Bitcoin, for example. In Bitcoin what matters is that you can at least connect to someone to download blocks and broadcast transactions. Or what about torrents? A swarm of nodes can download chunks of data and spread them between them. In other words: these systems don't require direct connectivity to every potential node for them to function. But what about the systems that do?

What if I'm trying to build a chat client and I'd like for two people to be able to share files between them? In this case I need both parties to be able to connect directly to each other. I have already spoken about how relying on a single method like port forwarding is prone to failure. The approach taken by P2PD is to combine multiple techniques for bypassing NATs and firewalls into one methodology which greatly improves the chances of success.

In the next section I will explain *How P2PD works*

1.2 How P2PD works

1.2.1 Adressesses

In P2PD it all starts with the address. You may already be familar with IPv4 and IPv6 addresses. Such addresses allow for data to reach nodes on the packet-switched network we call the Internet. Practically we can say that these addresses are assigned to routers. They work quite well for regular servers but in a peer-to-peer context much more information is needed to describe a peer's network setup. Here is what a peer's address looks like.

```
0,1-[0,8.8.8.8,192.168.21.21,58959,3,2,0]-0-zmUGXPOFxUBuToh
```

As you can see there are already pieces of information you may recognize. There is an IPv4 address and a private LAN IP that belongs to the NIC interface associated with that external IP. But what is the other information? Well, the address format includes the following details.

- **Signaling offsets.** Peers include a list of offsets for the MQTT servers to use in the settings file for signaling messages.
- **Interface offset.** Peers include a list of interfaces to listen on. The interface offset is referenced in protocol messages so the correct interface is used.
- **External IP.** The WAN IP associated with a given interface route.
- **Internal IP.** The private address associated with a route. For IPv6 this field will include a link local address. For IPv4 this will be a specific IP for the NIC.
- **Listen port.** The port used to listen on by the peer's main server.
- **NAT type.** The main type of NAT if the router for a route.
- **Delta type.** Information on a NAT's port mapping logic.
- **Delta value.** Information on any patterns found in a NAT's port mappings.
- **Node ID.** A random identifier assigned to the node. The identifier is subscribed to in MQTT to receive signal messages. More on this later.

The address format can describe multiple interfaces and address families. The maximum interface number is currently limited to 3 per address family. The importance of these addresses is they are what's used to open direct connections to a given peer. You will need to think about the best way to get these addresses when writing software. For example: are you going to assume people will give each other their addresses over a chat program? The next section will give you a better idea.

1.2.2 Signaling

P2PD uses the MQTT protocol for signaling messages. Signaling messages refer to messages P2PD uses to coordinate connections between peers. Some examples of such messages include:

- **P2P_DIRECT** = Tell a peer to connect back to an address.
- **ECHO** = Tell a peer to echo back data sent to it. Very useful for testing whether a system actually works!
- **INITIAL_MAPPINGS** = Exchange predicted port mappings as part of the sequence of events leading up to TCP hole punching.

Signaling messages are instrumental to the workings of P2PD. By relying on public, open, MQTT servers messages are able to reach peers directly without the cumbersome restrictions of a NAT. The way this occurs is through the use of random IDs as topic subscriptions. A peer subscribes to a random ID and includes this ID in its address information. Messages can then reach that peer via an MQTT broker server. Such an approach is scalable and already has a wide variety of public infrastructure.

You may be more familar with Bitcoin and how it initially used IRC to connect to its peer-to-peer network. What Bitcoin was doing was using IRC as a 'pub-sub' system. Specific channels were marked topics to subscribe to. Then the rooms were joined and channel members served as public entry points to the Bitcoin peer-to-peer network.

MQTT can be used in the same way. As a publish-subscribe system. But it's actually built for the purpose. Making it easier to use, more scalable, and far less hacky.

1.2.3 Methodology

P2PD uses 4 different strategies to try establish a connection between peers.

1. Direct Connect

If a peer has successfully port forwarded their main server then a regular TCP connection can be openned. There is nothing special about this.

2. Reverse Connect

In peer-to-peer connections both sides run a server. Therefore: a connection can be established if A connects to B or vice-versa. Reverse connect tells a peer to connect back to the node sending the request. This means that connectivity is successful if either one of two peers wishing to connect has been able to port forward.

3. TCP Hole Punching

There is a little known feature of TCP that allows for a connection to be openned if two sides connect to each other at the same time. You may be familar with a process called the 'SYN three-way handshake.' What this involves is the exchange of small message flags in order to open a new TCP connection. It so happens that if two sides connect to each other at the same time it's possible for these flag packets to arrive in such a way that it opens a new valid connection. This is *much* easier said than done.

For TCP hole punching to work certain conditions need to be met.

1. **Synchronicity** – Peers need to exchange packets at roughly 'the same time.' In distributed systems synchronizing peers is known to be a very difficult problem. P2PD uses the NTP protocol to achieve 1 - 30 ms accuracy.
2. **Latency** – The closer together two peers are from each other, the sooner packets will arrive, and the harder it is to do TCP hole punching. If packets in the SYN three-way handshake arrive too soon then the NAT can reject it; Latency and synchronicity are important.
3. **Predictability** – TCP hole punching relies on the ability to predict how a NAT will map the external ports used for outgoing connections. Many NAT types are highly predictable. Not all NATs exhibit properties that are predictable. These will fail TCP hole punching.
4. **Restrictiveness** – Some NATs impose special conditions in order for predictability to be preserved. E.G. requiring a certain reply port for an inbound connection. Some NATs are too restrictive. They will randomize all connections or impose restrictive firewalls.

P2PD's TCP hole punching feature has been tested on many different NAT configurations and operating systems. It can work behind NATs too to help bypass firewalls within a LAN. It also works inside virtual machines which seem to impose more restrictions on direct connectivity. Even so – TCP hole punching can still fail – and a last resort is needed.

4. TURN

TURN is a protocol that provides a generic proxy service for TCP and UDP traffic. It is utilized within WebRTC as a last resort approach for connecting peers when all other connection establishment options have failed. Since TURN servers must relay all traffic between peers it is much more expensive and centralized than other options. Hence why TURN is only used as a last resort.

In P2PD TURN support is not part of the default strategies for P2P connections as it utilizes UDP instead of TCP which would be inconsistent with other approaches. The TURN client I have implemented includes a feature that automatically acknowledges messages and retransmits them. Though sequencing has not been provided. The client is implemented in such a way that it provides an identical API to the connections returned from following any of the above strategies.

1.2.4 Next Steps

Now you have a good understanding how P2PD works. Choose a specialty:

1. **I want to learn *how to use the P2PD REST API.***
   I'm not interested in touching any Python code.
2. **I want to learn *how to use P2PD's library in my Python 3 program.***
   I think Python is le based so let's use it.

1.3 The P2PD REST API

1.3.1 Starting the server

Start the REST API server::
python3 -m p2pd.rest_api

Running this command will start the server on http://127.0.0.1:12333/ The server has no password and will only allow requests from an 'origin' of 127.0.0.1 or null. The null origin occurs when a HTML document is openned locally. If a website you visit tries to use the P2PD API your browser will include the domain name as an origin which the server will reject.

1.3.2 Making your first request

To check the server you can visit the version resource.

```
curl http://localhost:12333/version
```

You should see a JSON response.

```
{
   "author": "Matthew@Roberts.PM",
   "title": "P2PD",
   "version": "0.1.0"
}
```

1.3.3 Looking up your peer's address

You'll want to know how to get your peers address. The address is used to try connect to a peer.

```
curl http://localhost:12333/p2p/addr
```

Sample JSON response.

```
{
   "addr": "[0,1.3.3.7,192.168.21.21,58959,3,2,0]-0-looongcatislong",
   "error": 0
}
```

1.3.4 Now lets try connect to it

Peer addresses will be passed to the 'open' resource. A number of strategies are used to try establish connections. The order of success will define how fast connections can be openned.

```
curl "http://localhost:12333/p2p/open/name_for_new_con/addr_of_node"
```

Please note how the connection is given a name. The name is used to identify connections rather than using IDs. You will need to remember names for later API calls. Should you wish to test P2P connections you can also use 'self' to connect to yourself.

```
curl http://localhost:12333/p2p/open/con_name/self
```

The JSON response shows information on the new connection.


```
{
   "fd": 1060,
   "if": {
      "name": "Intel(R) Wi-Fi 6 AX200 160MHz"
      "offset": 0
   },
   "laddr": [
      "192.168.21.21",
      58537
   ],
   "name": "con_name",
   "raddr": [
      "192.168.21.21",
      58959
   ],
   "route": {
      "af": 2,
      "ext_ips": [
            {
               "af": 2,
               "cidr": 32,
               "ip": "1.3.3.7"
            }
      ],
      "nic_ips": [
            {
               "af": 2,
               "cidr": 32,
               "ip": "192.168.21.21"
            }
      ]
   },
   "strategy": "direct connect"
}
```


You can see the information includes details like the file descriptor number of the socket, your external address for the socket, and the technique that worked to establish the connection.

1.3.5 Text-based send and receive

Let's start with something simple. For these examples I'll assume you want to work with a simple text-based protocol. In reality you may be building something far more complex and require more flexibility but this is a good starting point.

Sending text:

The node server has a built-in echo server. We'll be using this protocol to test out some commands.

```
curl "http://localhost:12333/p2p/send/con_name/ECHO%20hello,%20world!"
```

```
{
   "error": 0,
   "name": "con_name",
   "sent": 18
}
```

Receiving text:

```
curl "http://localhost:12333/p2p/recv/con_name"
```

```
{
   "data": "hello, world!",
   "error": 0,
   "client_tup": [
      "192.168.21.200",
      54925
   ]
}
```

1.3.6 Binary send and receive

So far all API methods have used the GET method. GET is ideal for regular, text-based data where you don't have to worry too much about encoding. But if you want a more flexible approach that can also deal with binary data it's necessary to visit the POST method. These next examples will be written in Javascript using the Jquery library.

```
async function binary_push()
{
   // Binary data to send -- outside printable ASCII.
   // Will send an echo request to the Node server.
   var x = new Uint8Array(9);
   x[0] = 69; // 'E'
   x[1] = 67; // 'C'
   x[2] = 72; // 'H'
   x[3] = 79; // 'O'
   x[4] = 32; // ' '
   x[5] = 200; // ... binary codes,
   x[6] = 201;
   x[7] = 202;
   x[8] = 203;

   // Send as encoded binary data using POST to API.
   // This demonstrates that binary POST works.
```

(continues on next page)

(continued from previous page)

```
   var url = 'http://localhost/p2p/binary/con_name';
   var out = await $.ajax({
      url: url,
      type: "POST",
      data: x,
      contentType: "application/octet-stream",
      dataType: "text",
      processData: false
   });
}
```

```
{
   "error": 0,
   "name": "con_name",
   "sent": 9
}
```

Here's what it looks like to receive the binary back again.

```
async function binary_pull()
{
   // Receive back binary buffer.
   // Node server should echo back the last 4 bytes.
   var url = 'http://localhost/p2p/binary/con_name';
   out = await $.ajax({
      url: url,
      type: 'GET',
      processData: 'false',
      dataType: 'binary',
      xhrFields:{
         responseType: 'blob'
      },
      headers: { 'X-Requested-With': 'XMLHttpRequest' }
   });

   // Convert output blob to array buffer.
   // Then convert that to a Uint8Array.
   mem_view = await out.arrayBuffer();
   out_bytes = new Uint8Array(mem_view);
}
```

By the way: these examples use el8 async await syntax. This avoids callback hell which is low IQ. If you don't understand async-style code it's time for you to learn! All code in P2PD is async.

1.3.7 Bidirectional relay pipes

Theses simple send/receive calls are examples of push and pull APIs. In other words – its up to you to check whether messages are available. Such an approach might be fine for simple scripts but it's a little inefficent having to constantly check or 'poll' for new messages. Fortunately, P2PD has you covered. There is a special API method that converts a HTTP connection into a two-way relay.

What I mean by this is if you make a HTTP request to a named connection P2PD will relay data you send to that connection to the named connection and back again. This is very useful because it allows you to write asynchronous code that only has to handle data when it's available. Almost like a regular connection you made yourself.

The catch is I can't write the code for you exactly as I don't know what language you'll be using with the API – but so long as you know how to make a connection and send a HTTP request the process is quite straight-forwards.

1. Make a **SOCK_STREAM** socket. Choose **AF_INET** for the address family.
2. Connect the socket to **localhost** on port **12333**.
3. Send a HTTP GET request to /p2p/pipe/con_name. Data to send:

```
GET /p2p/pipe/con_name HTTP/1.1\r\n
Origin: null\r\n\r\n
```

The connection is closed on error. You can test it works by sending 'ECHO hello world' down the connection and checking for the response. As this is a relay between an associated connection to a peer's node server which implements echo.

1.3.8 Publish-subscribe

To learn about how to use the REST API for topic filtering please read the *Queues* page.

1.4 Using P2PD from Python

Before we get started all Python examples assume:

1. The 'selector' event loop is being used.
2. The 'spawn' method is used as the multiprocessing start method.
3. You are familar with how to run asynchronous code.

This keeps the code consistent across platforms. The package sets these by default so if your application is using a different configuration it may not work properly with P2PD.

Let's start with how to open a P2P connection to a peer.
We'll connect to ourselves for this example.

```
from p2pd import *

# Put your custom protocol code here.
async def msg_cb(msg, client_tup, pipe):
    # E.G. add a ping feature to your protocol.
    if b"PING" in msg:
```

(continues on next page)

(continued from previous page)

```
        await pipe.send(b"PONG")

async def make_p2p_con():
    # Initalize p2pd.
    netifaces = await init_p2pd()
    #
    # Start our main node server.
    # The node implements your protocol.
    node = await start_p2p_node(netifaces=netifaces)
    node.add_msg_cb(msg_cb)
    #
    # Spawn a new pipe from a P2P con.
    # Connect to our own node server.
    pipe = await node.connect(node.addr_bytes)
    pipe.subscribe(SUB_ALL)
    #
    # Test send / receive.
    msg = b"test send"
    await pipe.send(b"ECHO " + msg)
    out = await pipe.recv()
    #
    # Cleanup.
    assert(msg in out)
    await pipe.close()
    await node.close()

# Run the coroutine.
# Or await make_p2p_con() if in async REPL.
async_test(make_p2p_con)
```

You can use this library as a black box if you want. The code automatically handles loading network interfaces, enumerating routers, bypassing NATs, and establishing P2P connections. But you can do far more with P2PD.

It can be used as a way to do network programming in general. Whether you want to write multi-protocol, multi-address clients or servers. Using P2PD makes this simple. And it supports either using async or sync callbacks or a pull / push style API. Something like the Python equivalent of 'protocol classes' versus 'stream reader / writers' but with more control.

If you were to use Python by itself for network programming you would likely have to implement some of these features:

- **IPv6-specific socket bind code**
  - Link-local address logic
  - Global-address logic
  - Platform-specific logic
- **Interface-specific connection addresses**
  - Different for IPv6 link local addresses
  - Different for IPv6 global addresses

- Interface support (in general)
- External address suport
- **Whether to use 'protocol' classes or 'streams'**
  - Protocols = events; streams = async push and pull.
  - Python doesn't have an async push and pull API for UDP at all because Guido van Rossum thought it was a bad idea. I don't agree. P2PD can do async awaits on UDP. Or it can do event-based programming like the protocol class.
- Message filtering (useful for UDP protocols)
- Multiplexing-specific logic for UDP
- **Some very smart hacks to reuse message handling code**
  - Python has radically different approaches for TCP cons and servers.
  - While it does not provide the same methods for UDP.
  - I've created software that provides the same API features whether its a server or connection; TCP or UDP; IPv4 or IPv6

Fortunately I've done this already!
The next topics teach you more about network programming with P2PD.

1.4.1 Basics

Running async examples

Alr1ght people, P2PD uses Python's 'asynchronous' features to run everything in an event loop. You might want to use the special 'REPL' that the asyncio module provides to run these examples. It's available on (very) recent versions of Python like 3.8 or higher. Otherwise, P2PD has a function called async_test(name_of_async_func, arg_tup) that can be used to run async code.

```
python3 -m asyncio
```

```
asyncio REPL 3.11.0
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> from p2pd import *
>>> netifaces = await init_p2pd() # Loads netifaces.
```

Now you can simply type *await some_function()* in the REPL to execute it. If you experience errors in the REPL you'll have to use a regular Python file for the examples.

Network interface cards

All network programming in P2PD starts with the network interface card. Usually your computer will have a 'default' interface that all traffic is sent down based on various routing methods. Let's start by loading this default interface and interacting with it. Starting the interface looks up all its internal and external addresses, builds basic routing tables, and enumerates the NAT qualities of associated gateways.

```
# Start the default interface.
i = await Interface().start()

# Load additional NAT details.
# Restrict, random port NAT assumed by default.
await i.load_nat()

# Show the interface details.
repr(i)
```

```
Interface.from_dict({
    "name": "Intel(R) Wi-Fi 6 AX200 160MHz",
    "nat": {
        "type": 5,
        "nat_info": "restrict port",
        "delta": {
            "type": 6,
            "value": 0
        },
        "delta_info": "random delta (local port == rand port)"
    },
    "rp": {
        "2": [
            {
                "af": 2,
                "nic_ips": [
                    {
                        "ip": "192.168.21.21",
                        "cidr": 32,
                        "af": 2
                    }
                ],
                "ext_ips": [
                    {
                        "ip": "1.3.3.7",
                        "cidr": 32,
                        "af": 2
                    }
                ]
            }
        ],
        "23": []
    }
})
```

Repr shows a serializable dict representation of the interface after it's been loaded. You can see a list of interfaces available on your machine by using the **Interface.list()** function. Interfaces may be virtual, contain loopback devices,

and other adapters that aren't directly useful for networking. Often we are only interested in the adapters that are usable for WAN or LAN networking.

```
# Returns a list of Interface objects for Inter/networking.
netifaces = await init_p2pd()
ifs = await load_interfaces(netifaces=netifaces)
```

Now you know how to lookup interfaces and start them. It's time to learn about 'routes.'

The addressing problem

Modern network programming with event loops makes it incredibly easy to write high-performance networking code. The engineers of today are spoiled by such elegant features compared to the tools available in the early days. But there is still something very basic missing from the networking toolbelt:

The ability to easily know your external addressing information

There are many cases where this information is needed. For example imagine a server that listens on multiple IPs such that it is available on more than one external IP. The server may wish to know what external IPs are available to it in case it needs to refer a client to another server. The STUN protocol is an example of just this case where a client can request a connection back 'from a different IP address' in order to determine what type of NAT they have.

P2PD makes all external addressing information available to the programmer so that servers and clients can be aware of their own addresses.

Routes to the rescue

P2PD solves the addressing problem by introducing mappings called 'Routes' to describe how interface-assigned addresses relate to external addresses. Each route is indexed by address family. Either IPv4 or IPv6. A Route has the following basic form.

[NIC IPR, ...] -> [external IPR]

Example 1 – IPv4 routes

```
NIC IPs:
    192.168.0.20/32 (1 IP)
    193.168.0.0/16 (65024 IPs)
    7.7.7.7/32 (1 IP)
    8.8.0.0/16 (65024 IPs)

EXT IPs:
    1.3.3.7/32 (1 IP)
    8.8.0.0/16 (65024 IPs)

------------------------------------------------------------------
Routes:
    [...20, 193..., 7.7.7.7] -> [1.3.3.7]
    [8.8.0.0] -> [8.8.0.0]

Explanation:
    1.  The software starts by grouping all private addresses for a NIC.
        It then binds to one of the addresses and checks the external IP
        using STUN. The result is saved as the external address and this
        becomes a new route. When it finds a public IP for a NIC address
```

(continues on next page)

(continued from previous page)

```
        it binds to the first IP in it's range and checks the external
        IP. Here it finds that 7.7.7.7 results in the same external
        address as the other private IPs and groups them into the same
        route. This demonstrates that public IPs can be assigned to NICs
        and they don't necessarily mean that an IP is externally routable.

    2.  The software finds that when processing the block of IPs '8.8.0.0/16'
        that the external address matches. It assumes that this means the
        whole block is valid without checking every IP. This becomes another
        route. This example shows how some machines set their NIC IPs to
        their external addresses. It also demonstrates how ranges work.
```

Example 2 – IPv6 routes

```
NIC IPS:
    2020:DEED:BEEF::0000/128 (global scope) (1 IP)
    2020:DEED:DEED::0000/64 (global scope) (a lot of IPs)
    FE80:DEED:BEEF::0000/128 (link-local) (1 IP)

EXT IPS:
    2020:DEED:BEEF::0000/128 (global scope) (1 IP)
    2020:DEED:DEED::0000/64 (global scope) (a lot of IPs)

------------------------------------------------------------------
Routes:
    [FE80:DEED:BEEF::0000/128] -> [2020:DEED:BEEF::0000/128]
    [FE80:DEED:BEEF::0000/128] -> [2020:DEED:DEED::0000/64]

Explanation:
    1.  The algorithm for building routes in IPv6 is slightly different to IPv4.
        All link-local addresses for a list and are copied to the NIC
        section of the route. While every global addresss -- whether it's a
        single IP or a block -- creates a new route.
    2.  P2PD uses the EXT portion of routes for IPv6 servers. While it uses
        the NIC portion for IPv4. It is assumed that all servers should be
        publically reachable. Though this can be bypassed by specifying IPs
        directly for bind code which is indeed what the P2PD REST server does.
```

The reason why routes are important is they are used in bind() code to instruct what external addresses to use for servers or what external addresses will be visible for outbound traffic. In other words when you bind in P2PD you are selecting what external addresses to use.

A first networking program

Connect to Google.com and get a response from it.

```
from p2pd import *

async def main():
    #
    # Load default interface.
    i = await Interface().start()
```

(continues on next page)

(continued from previous page)

```
    #
    # Get first supported address family.
    # E.g. if the NIC only supports IPv4 this will == [AF_INET].
    # If it supports both families it will == [AF_INET, AF_INET6].
    # If no AF for route and address specified = use i.supported()[0].
    af = i.supported()[0]
    #
    # Get a route to use for sockets.
    # This will give you a copy of the first route for that address family.
    # Routes belong to an interface and include a reference to it.
    route = await i.route(af).bind()
    #
    # Lookup Google.com's IP address -- specify a specific address family.
    # Most websites support IPv4 but not always IPv6.
    # Interface is needed to resolve some specialty edge-cases.
    dest = await Address("www.google.com", 80, route).res()
    #
    # Now open a TCP connection to that the destination.
    pipe = await pipe_open(TCP, dest, route)
    #
    # Indicate that messages received should also be queued.
    # This enables the pull / push API.
    # You specify regex here. By default it will subscribe to everything.
    pipe.subscribe()
    #
    # Send it a malformed HTTP request.
    buf = b"Test\r\n\r\n"
    await pipe.send(buf)
    #
    # Wait for any message response.
    out = await pipe.recv(timeout=3)
    print(out)
    #
    # Cleanup.
    await pipe.close()

# From inside the async REPL.
await main()
```

You can see from this example that P2PD supports duel-stack networking, multiple network interface cards, external addressing, DNS / IP / target parsing, and publish-subscribe. But there are many more useful features for network programming.

In the next section we'll be taking a closer look at pipes.

1.4.2 Pipes

In P2PD all data messages are sent and received via pipes. Pipes are simply the name given to the object providing a common list of functions for transmission and message processing. A pipe supports UDP or TCP and IPv4 or IPv6.

```
async def pipe_open(proto, dest=None, route=None, sock=None, msg_cb=None, conf=NET_CONF):
    """
    route = Route object that's been bound with await route.bind().
    proto = TCP or UDP.
    dest  = If it's a client pipe a destionation should be included.
                await Address('host/ip', port, route).res()
            A server includes no destination.
    sock  = Used to wrap a pre-existing socket in a pipe. If the protocol is
            TCP and dest is included the socket is assumed to be connected.
    cb    = A message handler registered with servers before they're started
            so that messages aren't received before a handler is setup.
    conf  = A dictionary describing many different configuration options for
            changing various properties of the pipe.

    More details on msg_cb format and conf format later.

    Returns: a pipe object.
    """
```

Whether a pipe is for a client or server, UDP or TCP, IPv4 or IPv6, every pipe works the same. Pipes have been designed to process messages as they arrive. They pass these messages to any registered handlers (to process in real-time) or message queues (to be processed later).

TCP echo server example

Starts a simple TCP server that writes back received data down the client pipes for the sender. If this example works you should see nothing. Notice that msg handlers include a field for the senders addressing information and a pipe that can be used to interact with that client.

```
from p2pd import *

async def msg_cb(msg, client_tup, pipe):
    await pipe.send(msg, client_tup)

async def main():
    #
    # Start default interface and get the first route.
    # No AF for route use i.supported()[0]
    i = await Interface().start()
    route = await i.route().bind() # Port 0 = any unused port.
    #
    # Start the server and use msg_cb to process messages.
    server = await pipe_open(TCP, route=route, msg_cb=msg_cb)
    #
    # Connect to the server.
    # Use the IP of the route and unused port for the destination.
    dest = await Address(*server.sock.getsockname()[0:2], route).res()
    client = await pipe_open(TCP, dest, route)
```

(continues on next page)

(continued from previous page)

```
    client.subscribe()
    #
    # Send data to the server and check receipt.
    msg = b"test msg."
    await client.send(msg)
    out = await client.recv()
    assert(msg == out)
    #
    # Close both.
    await client.close()
    await server.close()

# From inside the async REPL.
await main()
```

UDP await example

In Python if you want to do asynchronous programming you're likely going to be writing different code for TCP and UDP. This is because TCP is 'stream-based' and UDP is 'packet-based.' TCP streams are reliable and ordered. UDP communication is not. So in Python for TCP connections you will be dealing with 'streams' while for UDP you will use protocol classes.

Only stream readers are 'asynchronous' e.g. you can await 'draining' a writer or await a reader - while there is no such equivalent for UDP. It's all very **inconvenient**. Wouldn't it be great if you could use asychnronous awaits for UDP and TCP? Further: wouldn't it be great if you modelled interactions in such a way that the same code would work for both?

Here's an example of how simple P2PD makes this. Here I'm using await for UDP which is based on message queues. Since there is no delivery guarantees for UDP it's possible this example throws a timeout error for you. Real-world code that deals with TCP usually has retransmissions built-in after a set duration. But no such logic here has been included. Note that the await for the recv is fully asychnronous. The event loop is free to run other tasks until a match occurs.

```
import random
import binascii
from p2pd import *

# Open default interface.
# Get a route for the first AF supported.
i = await Interface().start()
route = await i.route().bind()

# Open a UDP pipe to p2pd.net's STUN server.
# Subscribe to all messages.
pipe = (
    await pipe_open(
        UDP,
        await Address("p2pd.net", 34780, route).res(),
        route
    )
).subscribe()

# Build a STUN request and send it.
msg_id = ''.join([str(random.randrange(10, 99)) for _ in range(16)])
```

(continues on next page)

(continued from previous page)

```
req_hex = "00010000" + msg_id
req_buf = binascii.unhexlify(req_hex)
await pipe.send(req_buf)

# Get the response.
resp = await pipe.recv()
print(resp)
```

Pipe methods

Pipes are an instance of the BaseProto class that provides many useful methods and properties for working with connections (TCP or UDP.) Assume all of these methods are of the form 'pipe.method_name()' and that they 'belong' to a BaseProto class instance.

def subscribe(self, sub=SUB_ALL, handler=None)

Install a new message queue indexed by the regex pair sub = [msg_regex, client_tup_regex]. Doing this enables the use of publish-subscribe e.g. push / pull style awaits for a message. By default it will subscribe to all messages.

```
# Match any message containing meow.
# Allow only hosts from the 192.168.0.0/16 subnet.
# Put them into the same queue.
sub = [b"meow", b"192[.]168[.][0-9]+[.][0-9]+:[0-9]+"]
pipe.subscribe(sub)

# Wait for a message that fits into the sub queue.
await pipe.recv(sub, timeout=4)
```

def unsubscribe(self, sub)

Delete the queue and its resources marked by sub (if it exists.) No longer copy messages that fit this subscription into this queue.

async def recv(self, sub=SUB_ALL, timeout=2, full=False)

Given a queue identified by the subscription 'sub' – wait for a message that suites it. Waiting is done asychnronously and other tasks may be done by the event loop until a message arrives. Timeout specifies the total duration to attempt to wait. After the duration an exception will be thrown. Set this to 0 to disable timeouts (not recommended.)

```
# Wait for any message from a loopback client.
out = await pipe.recv([b"[\s\S]+", "127.0.0.1:[0-9]+"])
```

By default this function only returns the message received on the pipe. Some pipes receive messages from multiple destinations (like UDP.) To also show the sender set the full flag to True. The return value will end up being [msg_bytes, client_tup].

async def send(self, data, dest_tup=None)

Wait for data to be trasmitted down the pipe (non-blocking.) For TCP / UDP connections (with a fixed destination) the dest_tup does not need to be set. But it's a good practice to include it in servers because the same socket in UDP servers is used to receive messages from multiple clients and the pipe by itself won't be able to disambiguate what the destination should be. This is also one reasons why msg_cbs include a client_tup for a message sender.

def add_msg_cb(self, msg_cb)

When a pipe receives a message it will also forward it to any installed message handlers. The format for a message handler is:

async def msg_cb(msg, client_tup, pipe)

The msg_cb also doesn't have to be an async callback but keep in mind if it's given as a regular function you will have to use asyncio.create_task to schedule any callbacks and you won't be able to await them. Since the whole library uses async await it's best just to use an async msg_cb.

Using message handlers like this is very useful because you can install them for either a server pipe or a client pipe and it will automatically be called when there's a new message. No need to run your own loop and call awaits on some object. The event loop handles it.

def del_msg_cb(self, msg_cb)

Removes a function reference designated by msg_cb from the pipe's msg_cbs.

def add_end_cb(self, end_cb)

When a connection is closed manually or forcefully the end_cb handlers are called. These are useful for cleanup. The format is:

async def end_cb(msg, client_tup, pipe)

Where message is set to None.

def del_end_cb(self, end_cb)

Removes a function reference designated by end_cb from the pipe's end_cb handlers.

def add_pipe(self, pipe)

Pipes can be made to route messages to other pipes. You can connect two pipes together by adding each pipe to each other.

```
pipe_a.add_pipe(pipe_b)
pipe_b.add_pipe(pipe_a)
```

1. Messages received at pipe_a will be sent down pipe_b.
2. Messages received at pipe_b will be sent down pipe_a.

This doesn't cause looping as the messages get sent to the destination rather than the pipe itself. Linking pipes together is the trick used in the P2PD REST API for 'converting' an active HTTP connection into a two-way relay to an active P2P connection in only two lines of code.

def del_pipe(self, pipe)

Unlink 'pipe' from self.

async def close(self)

Closes all resources associated with a pipe. If it's a server it will stop serving any clients and all client connections will be closed. All sockets will be closed forcefully. Server's that immediately reuse the same port may experience errors where they fail to receive designated packets. There may be a solution to this by setting SO_LINGER to enabled and using a zero timeout. But using this option on client TCP sockets on Windows prevents the hole punching algorithm from working so this needs to only be considered for server sockets.

Additional pipe options

A default dictionary of configuration options is passed to each pipe. The options look like this:

```
NET_CONF = {
    # Only applies to TCP.
    "con_timeout": 2,

    # No of messages to receive per subscription.
    "max_qsize": 1000,

    # Require unique messages or not.
    "enable_msg_ids": 0,

    # Number of message IDs to keep around.
    "max_msg_ids": 1000,

    # Reuse address tuple for bind() socket call.
    "reuse_addr": False,

    # Setup socket as a broadcast socket.
    "broadcast": False,

    # Buf size for asyncio.StreamReader.
    "reader_limit": 2 ** 16,

    # Return the sock instead of the base proto.
    "sock_only": False,

    # Disable closing sock on error.
    "no_close": False,

    # Whether to set SO_LINGER. None = off.
    # Non-none = linger value.
    "linger": None,
```

(continues on next page)

(continued from previous page)

```
    # Ref to an event loop builder.
    "loop": None
}

# Here's where to use these options.
pipe = pipe_open(TCP, dest, route, conf=NET_CONF)
```

1.4.3 Queues

If you want to use the push and pull APIs over the relay pipes there are some additional details that you might find useful. In order to support pull API usage the software must be able to save messages. It does this by using in memory queues which for now have no set limit. What hasn't been mentioned is that these queues are organized around subscriptions – regex patterns that match messages and remote peers.

These queues only exist when a subscription is made. By default P2PD does not subscribe to anything when used as a library. But the REST API subscribes to 'any message' from 'any peer.' This has the special format of a blank message pattern and a blank peer address pattern to match everything. The reason why this feature exists is because of the way UDP is designed.

(Disclaimer: UDP really sucks.)

You may know that UDP offers no ordered delivery or indeed any kind of reliable delivery guarantee at all. In practice this means that UDP-focused protocols (like STUN) end up using randomized IDs in requests and responses as kind of an asynchronous form of 'ordering.' There is also the case that UDP is 'connectionless.' This means that you can have a single socket that you can use to send packets to multiple destinations.

What ends up happening is you get back messages [on the same socket] that:

1. **… Are from multiple different hosts and or ports.**
2. **… Are from multiple different requests.**

And it's just a mess. So I had the idea of being able to sort messages and remote (IP + port) tuples using regex. Such an approach is flexible enough for any kind of protocol and is already in use in my STUN client. Now here's what that looks like in practice. First for API then Python.

Javascript subscription example

```
en = encodeURIComponent;
async function p2pd_test(server)
{
    // Do these in order to test some P2PD APIs.
    msg_p = en("[hH]e[l]+o");
    addr_p = en("[\s\S]+");
    var paths = [
        "/version",
        "/p2p/open/con_name/self",
        "/p2p/sub/con_name/msg_p/" + msg_p + "addr_p" + addr_p,
        "/p2p/send/con_name/" + en("ECHO Hello, world!"),
        "/p2p/recv/con_name/msg_p/" + msg_p + "addr_p" + addr_p,
    ];
```

(continues on next page)

(continued from previous page)

```
    // Make requests to the API.
    for(var i = 0; i < paths.length; i++)
    {
        // Make API request.
        url = 'http://localhost:12333' + paths[i];
        var out = await $.ajax({
            url: url,
            type: 'GET',
            dataType: "text"
        });

        console.log(out);
    }
}
```

```
// Subscribe.
{
    "error": 0,
    "name": "con_name",
    "sub": "[b'[hH]e[l]+o', b'[\s\S]+']"
}

// Send data.
{
    "error": 0,
    "name": "con_name",
    "sent": 18
}

// Receive data.
{
    "data": "Hello, world!",
    "error": 0,
    "client_tup": [
        "192.168.21.200",
        54925
    ]
}
```

The URL encode method is used to make the data 'safe' to pass in a URL. A subscription consists of two regex patterns. The first regex matches a message while the second matches an 'IP:port'. Message queues are assigned to each subscription. When receiving messages from a queue the full subscription / regex pair must be included. In the example above a message pattern matches hello, Hello, helo, or Hello. The regex method is 'find_all' so any instance of the pattern returns a match. But you can always use the caret ^ and dollar $ characters to match a whole string:

```
Checkout https://regex101.com/ if you need help with your regexes!
```

Python subscription example

For brevity I won't go into using the library in this section. This is just an example to get a sense of what subscriptions look like from Python code.

```
from p2pd import *

async def sub_example():
    #
    # Start default interface.
    # Don't bother resolving external addresses.
    i = await Interface().start()
    #
    # Echo server address.
    route = await i.route().bind()
    echo_dest = await Address("p2pd.net", 7, route).res()
    #
    # Open a connection to the echo server.
    pipe = await pipe_open(TCP, echo_dest, route)
    #
    # Create a new queue for a related subscription.
    # Subscription = [b"msg pattern regex", b"address pattern regex"]
    # SUB_ALL = [b"", b""] meaning 'any message', 'from any address.'
    pipe.subscribe(SUB_ALL)
    #
    # Send data down the pipe.
    msg = b"do echo test\r\n"
    await pipe.send(msg, echo_dest.tup)
    #
    # Receive data back.
    data = await pipe.recv(SUB_ALL, 4)
    self.assertEqual(data, msg)
    #
    # Close the sockets.
    await pipe.close()

# Utility function to run an async function.
async_test(sub_example)
```

Last words on queues

What you should understand about subscriptions and queues is messages are delivered to all matching subscription queues. So if you subscribe to SUB_ALL / any message and a more specific subscription you will end up with copies of every message on the ALL queue with only the matching messages on the second one. You may only be interested in a specific message but if you subscribe to everything it will mean these messages are still duplicated there. So you may have to flush messages you've already processed should you want to use that queue.

The way to unsubscribe is to use the delete method.

```
curl -X DELETE "http://localhost:12333/p2p/sub/con_name/msg_p/regex/addr_p/regex"
```

```
async function p2pd_test(server)
{
```

(continues on next page)

(continued from previous page)

```
    var out = await $.ajax({
        url: "http://localhost:12333/p2p/sub/con_name/msg_p/regex/addr_p/regex",
        type: 'DELETE',
        dataType: "text"
    });

    console.log(out);
}
```

```
{
    "error": 0,
    "name": "con_name",
    "unsub": "[b'regex', b'regex']"
}
```

By default the msg_p and addr_p are set to blank if they're not included. Therefore to unsubscribe from 'all messages' don't include them.

```
curl -X DELETE "http://localhost:12333/p2p/sub/con_name"
```

```
{
    "error": 0,
    "name": "con_name",
    "unsub": "[b'', b'']"
}
```

1.4.4 Daemons

In the *Pipes* section I showed some simple examples of how servers can be built using the *pipe_open* function. Such an approach is fine if you only want to use one protocol or address type. But real-world servers may need to run on multiple routes, address, and interfaces. The Daemon class offers a way to build such servers.

To use it you sub class it and define your own msg_cb function.

listen_all

```
from p2pd import *

class EchoServer(Daemon):
    def __init__(self):
        super().__init__()
    #
    async def msg_cb(self, msg, client_tup, pipe):
        await pipe.send(msg, client_tup)

async def main():
    i = await Interface().start()
    #
    # Daemon instance.
    server_port = 10126
```

(continues on next page)

(continued from previous page)

```
    echod = await EchoServer().listen_all(
        [i],
        [server_port],
        [TCP]
    )
    #
    await echod.close()

await main()
```

The listen_all method of the Daemon class is as follows.

```
async def listen_all(self, targets, ports, protos, af=AF_ANY, msg_cb=None,  error_on_
↪af=0)
```

Targets, ports, and protos are a list. The supported objects that can be used as targets are Interfaces and Routes. A range may be given as an entry in the ports list by using [start, stop] (inclusive.) The total number of servers created will be len(targets) * total_ports * len(protos) * total_afs. For example:

```
targets = [i, r]
ports = [10000, [30000, 30100]]
protos = [TCP]
af = AF_ANY (use all supported address families of the related interface.)
```

Let's assume that i and r use duel-stack interface. The total servers would be 2 * 101 * 1 * 2 = 404 so it can add up fast if you're not careful. It should be noted that any Route objects passed as targets will have their bound information ignored if they're already bound. But their NIC IPs and EXT IPs will still be used as a template to create routes for the servers based on the other parameters used for listen_all.

Note: IPv6 note: P2PD has small differences in the way it handles IPv6 services compared to IPv4 which would be unexpected without learning about them. In IPv4 to run a server on a network interface that can be reached from the LAN and WAN you simply listen on one of the NICs IPs. In IPv6 it has kind of split up LAN IPs and external IPs into link-local addresses and global-scope addresses, respectively.

I wanted the code to work the same with IPv6 so I start with the assumption that users want servers to be reachable internally and externally. Hence I don't just bind to a global-scope address. I also bind to a link-local address. That means that for IPv6 the final number of servers created is multipled by 2. Maybe this is a bad idea but I think it simplifies a lot of things.

---

listen_specific

The listen_all function is useful for applying the same AFs, protocols, and ports to the entries in the targets list. But sometimes you want to use the targets as-is if they're already 'bound.' Perhaps in the case where they've been specifically set to bind to a loopback adapter or even 'all addresses.'

```
async def listen_specific(self, targets, msg_cb=None)
```

The format of targets here is given as **[[target, protocol], ...]**.

```
p = 10233
d = Daemon()
```

(continues on next page)

(continued from previous page)

```
i = await Interface().start()
b = await i.route(i.supported()[0]).bind(ips="127.0.0.1")
await d.listen_specific(
    targets=[[b, TCP]],
)

await d.close()
```

The listen_specific code hasn't been tested too much so it's better to use **listen_all**.

1.4.5 TURN client

TURN is a protocol used for relaying TCP and UDP end-points between peers. In P2PD I've implemented an asynchronous, IPv4 / IPv6, TURN client. It has the same interface as a pipe and supports awaits + callbacks. But the TURN client uses UDP over TCP. It provides reliable delivery but it does not provide ordered delivery (yet?)

The reason why UDP was implemented and not TCP is due to the way TCP works and what TURN is designed to be used for. P2PD already implements **direct connect, reverse connect, and TCP hole punching**. If all of these fail it means that there is very little chance of establishing a TCP connection with a peer. But believe it or not: this is the assumption made in the TURN protocol. The TURN server makes an outgoing connection to a service (which must be reachable.)

The TURN spec mentions that this could be combined with **TCP hole punching for clients**. But this is not feasible because no synchronization mechanism for making connections has been offered in the TURN protocol – at least none that I've seen. It would have been unlikely to work given that punching needs to be synchronized down to the milisecond. All of this could have been avoided if TURN were designed around reverse connections. So that both parties behind NATs could simply connect to the server and setup channels. But TURN doesn't seem to offer this possibility.

Note: Fun fact: **TURN is the worst protocol I've ever had the displeasure of working with**. It manages to make a simple proxy server look like a moonlanding mission. I don't even know how they managed to over-engineer the protocol to such a high-level. I don't think I could manage to fuck something up that badly even if I was trolling. True story.

UDP is a little better

When you go through the TURN protocol as a client you get allocated a special 'relay address' from the server that another peer can use to route messages to you. As far as I know this only works for UDP. But importantly it offers a reverse connect design which is capable of bypassing NATs.

UDP is a better choice as a last resort because it is 'connectionless.' It doesn't require the receipt of a handshake. The NAT will simply let through packets to a UDP socket as long as that socket address has already sent data to the destination. So UDP hole punching is easier than TCP.

The downside is… it's UDP. It offers no reliable delivery or sequencing. But a few hacks add delivery back in. It wouldn't be possible to add sequencing, too. Though I have not done this for now. That was a lot of text so let's look at some code.

```
from p2pd import *

# Network interface details.
i = await Interface().start()
```

(continues on next page)

(continued from previous page)

```
r = await i.route().bind()

# Address of a TURN server.
dest = await Address(
    "p2pd.net",
    3478,
    r
).res()

# Sync message callback -- do something here if you like.
# Can be async too.
msg_cb = lambda msg, client_tup, pipe: print(msg)

# Implement the TURN protocol for UDP send / recv.
client = TURNClient(
    turn_addr=dest,
    turn_user=None,
    turn_pw=None,
    turn_realm=b"p2pd.net",
    route=r,
    msg_cb=msg_cb
)

# Wait for authentication and relay address allocation.
await async_wrap_errors(
    client.start()
)
client.subscribe(SUB_ALL)

# Send a message to ourselves.
await client.send(b"hello, world!", await client.relay_tup_future)
# You should see the message printed.

# Alternatively the pub-sub API is available since we called subscribe.
out = await client.recv()
print(out)

# Cleanup.
await client.close()
```

Using TURN as a fall-back option

In P2P pipes there are three main methods used to establish a connection. All of these methods use TCP. By default TURN is disabled as a method for establishing a 'connection.' The reason for this is it does not provide ordered delivery like TCP does which might come as a shock to most developers. However, it can be enabled should a developer choose to use it.

```
from p2pd import *

async def make_p2p_con():
    # Initalize p2pd.
```

(continues on next page)

(continued from previous page)

```
    netifaces = await init_p2pd()
    #
    # Start our main node server.
    # The node implements your protocol.
    node = await start_p2p_node(netifaces=netifaces)

    # Strategies used to make a P2P connection.
    # Note that P2P_RELAY enables TURN.
    strategies = [ P2P_DIRECT, P2P_REVERSE, P2P_PUNCH, P2P_RELAY ]

    """
    Spawns a new pipe from a P2P connection.
    In this case it's connecting to our own node server.
    There will be no barriers to do this so this will just use
    a plain direct TCP connection / P2P_DIRECT.
    Feel free to experiment with how it works.
    """
    pipe = await node.connect(node.addr_bytes, strategies)

    # Do some stuff on the pipe ...
    # Cleanup.
    await pipe.close()
    await node.close()

# Run the coroutine.
# Or await make_p2p_con() if in async REPL.
async_test(make_p2p_con)
```

1.4.6 STUN client

STUN is a protocol that can be used to lookup a peer's external address. It also provides information on port mappings and some servers with multiple addresses support the optional feature of receiving connections back from another address. Consequently, such servers allow one to determine the type of NAT you're behind (if any.)

There are many public STUN servers that can be used for basic functionality. But very few support TCP, IPv6, or multiple addresses, and the ones that do are very unreliable and often experience downtime. P2PD uses STUN heavily to determine the routes that belong to interfaces and their NAT information. The only way to ensure the software worked reliability was to run a STUN server with multiple IPv4/6s on TCP and UDP.

Here is how to use the STUN client.

```
from p2pd import *

i = await Interface().start()

# Test echo server with AF.
stun_client = STUNClient(i, i.supported()[0])
wan_ip = await stun_client.get_wan_ip()
nat = await stun_client.get_nat_info()
results = await stun_client.get_mapping(TCP)
```

1.4.7 Examples

Here is a collection of example(s) that use P2PD in some form.

REST APIs

There exists various Python frameworks for building REST APIs. But what many people don't realize is the Python standard library already supports enough of HTTP that a library isn't often isn't required. Python is able to parse HTTP requests and responses into objects that are quite easy to use - though the location of these features in the standard library isn't that intuitive.

I've built a simple module that accesses these features. Using this module you can now easily build a TCP / UDP / IPv4 / IPv6 async REST API in Python. Here's an example of that.

```
from p2pd import *

class NetInfoServer(Daemon):
    async def msg_cb(self, msg, client_tup, pipe):
        # Parse HTTP message and handle CORS.
        req = await rest_service(msg, client_tup, pipe)
        #
        async def get_response():
            # Show information of the connection to the peer.
            p = req.api("/mapping")
            if p:
                sock = pipe.sock
                return {
                    "error": 0,
                    "fd": sock.fileno(),
                    "laddr": sock.getsockname(),
                    "raddr": sock.getpeername(),
                    "route": pipe.route.to_dict(),
                    "if": {
                        "name": pipe.route.interface.name
                    }
                }
            #
            # Fallback and serve all other purposes.
            return {
                "error": 2,
                "msg": "method not implemented"
            }
        #
        resp = await get_response()
        if resp is not None:
            await send_json(
                resp,
                req,
                client_tup,
                pipe
            )

async def start_server():
```

(continues on next page)

(continued from previous page)

```
    netifaces = await init_p2pd()
    #
    # Default interface of your machine.
    # netifaces.interfaces() for names
    # or await load_interfaces() for a started list.
    i = await Interface(netifaces=netifaces).start()
    #
    # Server object inherits from a standard Daemon.
    server = NetInfoServer()
    #
    # Makes a Route aware of all Routes used for the server.
    # Might do this automatically in the future.
    server.set_rp(i.rp)
    #
    # Defines addresses and protocols to listen on.
    # Feel free to switch this up.
    await server.listen_all(
        # Listen on all routes for IP4 and IPv6.
        [i.rp[IP4], i.rp[IP6]],

        # Port(s) to listen on.
        [20000],

        # Interested in TCP and UDP.
        [TCP, UDP]
    )
    #
    while 1:
        await asyncio.sleep(10)

async_test(start_server)
```

1.5 Future work

1.5.1 Network changes

What most production software has is some kind of logic to detect network changes. Events like disconnects, external IP changes, changes to the gateway, or active NIC, allow the program to react to changes. I'd like to eventually have code to detect such changes on all major operating systems. Then give pipes the ability to add handlers for such events. This would make it very easy to engineer highly reliable networking software.

1.5.2 Firewall bypass

Windows-based operating systems include firewalls that will prompt the user if they want to allow an application to use the Internet. Mac OS X has similar security measures. A user without much experience may not know what to do. It would be a good idea to automatically setup rules to allow P2PD to use the Internet on various platforms. I'd also like to put some work into packaging in the future so that this software is as easy to use as possible.

1.5.3 Pushing sockets

One common criticism against Python is that 'it's slow.' I don't believe this is the case but let's suppose that it is. Let's suppose that an engineer has a very good reason not to use something like P2PD for their peer-to-peer networking code e.g. they may have already written highly optimized networking code themselves. Well, it's possible to pass a socket from one process to another. What this could mean is that P2PD could specialize in the initial process of openning up connections with remote peers and then passing those bound sockets to other processes to use as they see fit.

I think this could be a really cool option because it would allow engineers to reuse their existing networking code. Maybe they have a different event loop. Maybe they use a model based on threads and polling. They would be able to use the networking features they're already familar with for their respective software stacks. I think it's an interesting idea.

1.5.4 Error recovery code

As I sit here reflecting on this project I'm reminded of just how many ways networking code can fail versus regular algorithms. As an example: on Windows if you switch between wireless networks there can be a delay until being able to use that interface for Internet traffic. I don't know why that is. It may be an issue with router advertisements and ARP. But what I know is any code that runs shortly after the network is changed is likely to fail despite having 'correct' addressing information.

I think it would be worthwhile making a list of common failure scenarios and writing code to prevent it from occuring. Really only the most basic networking features are provided in programming languages. There are many other ways sessions can fail and most developers manually handle the edge-cases themselves (like reconnect.)

1.5.5 Other ideas

1. Send duplicate signaling msgs in case a MQTT server goes down.
2. Ability to restart broken TCP connections after disruptions in Internet. Many simple servers start a fresh state per connection and in some scenarios (like multiplayer games – it can mean being unable to rejoin sessions.)