IPv8 Documentation

Release 2.12.0

Tribler

Oct 07, 2023

PRELIMINARIES:

1 Why does IPv8 exist? **3**

2 IPv8 Features **5**

3 The science behind IPv8 **7**

4 IPv8 Example **9**

5 Getting help **11**

6 Table of contents **13**

6.1 Installing Libsodium (Windows and MacOS only) 13
6.2 Creating your first overlay 13
6.3 Storing states in IPv8 19
6.4 Unit Testing Overlays 22
6.5 Task Management 30
6.6 Network IO and the DiscoveryStrategy 34
6.7 Using the IPv8 attestation service 36
6.8 Peer discovery basics 40
6.9 Community Best Practices 41
6.10 IPv8 configuration options 43
6.11 IPv8 bootstrapping 45
6.12 Key generation options 47
6.13 Message serialization 47
6.14 Advanced attestation service usage 56
6.15 Advanced peer discovery configuration 58
6.16 TunnelCommunity 60
6.17 Attestation Prototype Documentation 61
6.18 Using the IPv8 attestation service 70

7 Search **75**

Welcome to the IPv8 documentation! This file will help you navigate the existing documentation. The documentation will explain the concepts behind IPv8, the architecture, and guide you through making your first overlay.

CHAPTER

ONE

WHY DOES IPV8 EXIST?

Problems with the very fabric of The Internet, IPv4, are mounting. The approach of IPv6, Mobile IP, and IPSec is hampered by fundamental architectural problems. One solution is moving the intelligence up to a higher layer in the protocol stack and towards the end points. Our endpoints are not dependent on any central infrastructure. Our architecture is academically pure and fully decentralized; to the point of obsession. IPv8 is based on the principle of self-governance. Like Bitcoin and BitTorrent our IPv8 overlays are permissionless and requires no server upkeep costs.

IPv8 aims to help restore the original Internet; for free; owned by nobody; for everyone.

CHAPTER

TWO

IPV8 FEATURES

IPv8 is a networking layer which offers identities, communication with some robustness, and provides hooks for higher layers. For instance, our Internet-deployed reputation functions and ledger-based storage of reputation data. IPv8 is designed as a mechanism to build trust. Each network overlay offers network connections to known digital identities through public keys. Overlays are robust against several network problems and security issues. Using a custom NAT-traversing DHT to find the current IPv4 network address, IPv8 keeps the network connectivity going, even as the IPv4 addresses change. Each network overlay keeps track of a number of neighbors and occasionally checks if they are still responsive.

IPv8 offers global connectivity through integrated UDP NAT puncturing, announcement of your identity claim and a web-of-trust. IPv8 has an integrated attestation service. You can use IPv8 for official verification that something is true or authentic, according to a trustworthy attestor. By using zero-knowledge proofs we attempt to minimize privacy leakage.

CHAPTER

THREE

THE SCIENCE BEHIND IPV8

IPv8 was built through years of experience and was shaped by science. Some key publications are:

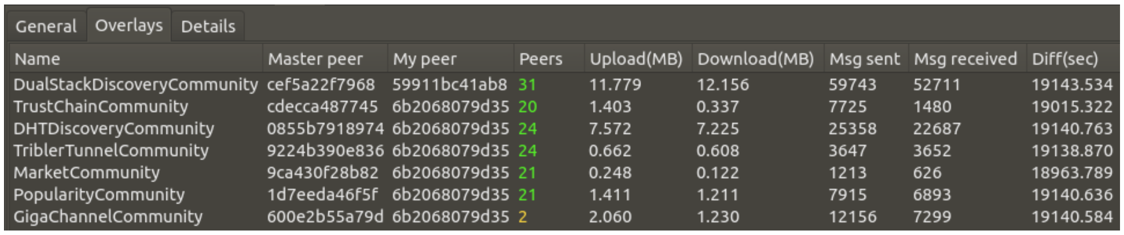
- Halkes G, Pouwelse J. UDP NAT and Firewall Puncturing in the Wild. In International Conference on Research in Networking 2011 May 9 (pp. 1-12). Springer, Berlin, Heidelberg.
- Zeilemaker N, Schoon B, Pouwelse J. Dispersy bundle synchronization. TU Delft, Parallel and Distributed Systems. 2013 Jan.

CHAPTER

FOUR

IPV8 EXAMPLE

IPv8 is a tool to build interesting distributed applications. IPv8 overlays can be used to offer various services for your application. Our flagship application Tribler, for example, uses seven IPv8 overlays for serverless discovery. Tribler's services range from DHT-based lookup, Tor-inspired privacy to a completely decentralised marketplace for bandwidth. If we take a look at Tribler's debug screen, it shows IPv8 in action: you can observe statistics such as active neighbors and utilized network traffic to make sure you have made a healthy overlay.



Name	Master peer	My peer	Peers	Upload(MB)	Download(MB)	Msg sent	Msg received	Diff(sec)
DualStackDiscoveryCommunity	cef5a22f7968	59911bc41ab8	31	11.779	12.156	59743	52711	19143.534
TrustChainCommunity	cdecca487745	6b2068079d35	20	1.403	0.337	7725	1480	19015.322
DHTDiscoveryCommunity	0855b7918974	6b2068079d35	24	7.572	7.225	25358	22687	19140.763
TriblerTunnelCommunity	9224b390e836	6b2068079d35	24	0.662	0.608	3647	3652	19138.870
MarketCommunity	9ca430f28b82	6b2068079d35	21	0.248	0.122	1213	626	18963.789
PopularityCommunity	1d7eeda46f5f	6b2068079d35	21	1.411	1.211	7915	6893	19140.636
GigaChannelCommunity	600e2b55a79d	6b2068079d35	2	2.060	1.230	12156	7299	19140.584

CHAPTER FIVE

GETTING HELP

If you spotted an inconsistency in the documentation or found a bug, please let us know in a GitHub issue.

CHAPTER

SIX

TABLE OF CONTENTS

6.1 Installing Libsodium (Windows and MacOS only)

Running py-ipv8 on Windows or MacOS, requires manual installation of Libsodium.

6.1.1 Windows

1. Libsodium can be downloaded from https://download.libsodium.org/libsodium/releases/

   ```
   For eg. https://download.libsodium.org/libsodium/releases/libsodium-1.0.17-msvc.zip
   ```

2. Extract the files from the zip file
3. There are two extracted directories: `x64` and `Win32`. Select `x64` for 64-bit or `Win32` for 32-bit versions of Windows, and search for `libsodium.dll`. You can find one inside `Release/v141/dynamic/libsodium.dll`
4. Copy this `libsodium.dll` file and paste it in `C:\Windows\system32`

6.1.2 MacOS

Homebrew can be used to install libsodium:

```
brew install libsodium
```

For details, check here.

6.2 Creating your first overlay

This document assumes you have installed all of the dependencies as instructed in the README.md. You will learn how to construct a *network overlay* using IPv8.

6.2.1 Running the IPv8 service

Fill your `main.py` file with the following code:

```
from asyncio import run

from ipv8.configuration import get_default_configuration
from ipv8.util import run_forever
from ipv8_service import IPv8


async def start_ipv8() -> None:
    # Create an IPv8 object with the default settings.
    ipv8 = IPv8(get_default_configuration())
    await ipv8.start()

    # Wait forever (or until the user presses Ctrl+C)
    await run_forever()

    # Shutdown IPv8. To keep things simple, we won't stop IPv8 in the remainder of the␣
↪tutorial.
    await ipv8.stop()

# Create a new event loop and run a task that starts an IPv8 instance.
run(start_ipv8())
```

You can now run this file using Python as follows:

```
python3 main.py
```

You should see some debug information being printed to your terminal. If this step failed, you are probably missing dependencies.

If everything is running correctly: congratulations! You have just run the IPv8 service for the first time.

6.2.2 Running two IPv8 services

Now that we have managed to create an IPv8-service instance, we want to create a second instance. This way we can start testing the network overlay with multiple instances. To try this, fill your `main.py` file with the following code:

```
from asyncio import run

from ipv8.configuration import get_default_configuration
from ipv8.util import run_forever
from ipv8_service import IPv8


async def start_ipv8() -> None:
    # The first IPv8 will attempt to claim a port.
    await IPv8(get_default_configuration()).start()
    # The second IPv8 will attempt to claim a port.
    # It cannot claim the same port and will end up claiming a different one.
    await IPv8(get_default_configuration()).start()
```

(continues on next page)

(continued from previous page)

```
    await run_forever()


run(start_ipv8())
```

If you were successful, you should now see double the debug information being printed to your terminal.

6.2.3 Loading a custom overlay

Now that we can launch two instances, let's create the actual network overlay. To do this, fill your `main.py` file with the following code:

```
import os
from asyncio import run

from ipv8.community import Community
from ipv8.configuration import ConfigBuilder, Strategy, WalkerDefinition, default_
↪bootstrap_defs
from ipv8.util import run_forever
from ipv8_service import IPv8


class MyCommunity(Community):
    # Register this community with a randomly generated community ID.
    # Other peers will connect to this community based on this identifier.
    community_id = os.urandom(20)


async def start_communities() -> None:
    for i in [1, 2]:
        builder = ConfigBuilder().clear_keys().clear_overlays()
        # If we actually want to communicate between two different peers
        # we need to assign them different keys.
        # We will generate an EC key called 'my peer' which has 'medium'
        # security and will be stored in file 'ecI.pem' where 'I' is replaced
        # by the peer number (1 or 2).
        builder.add_key("my peer", "medium", f"ec{i}.pem")
        # Instruct IPv8 to load our custom overlay, registered in _COMMUNITIES.
        # We use the 'my peer' key, which we registered before.
        # We will attempt to find other peers in this overlay using the
        # RandomWalk strategy, until we find 10 peers.
        # We do not provide additional startup arguments or a function to run
        # once the overlay has been initialized.
        builder.add_overlay("MyCommunity", "my peer",
                            [WalkerDefinition(Strategy.RandomWalk,
                                              10, {'timeout': 3.0})],
                            default_bootstrap_defs, {}, [])
        await IPv8(builder.finalize(),
                   extra_communities={'MyCommunity': MyCommunity}).start()
    await run_forever()
```

(continues on next page)

(continued from previous page)

```
run(start_communities())
```

As we replaced the default overlays, you should no longer see any debug information being printed to your terminal. Our overlay is now loaded twice, but it is still not doing anything.

6.2.4 Printing the known peers

Like every DHT-based network overlay framework, IPv8 needs some time to find peers. We will now modify `main.py` again to print the current number of peers:

```
import os
from asyncio import run

from ipv8.community import Community
from ipv8.configuration import ConfigBuilder, Strategy, WalkerDefinition, default_
↪bootstrap_defs
from ipv8.peerdiscovery.network import PeerObserver
from ipv8.types import Peer
from ipv8.util import run_forever
from ipv8_service import IPv8


class MyCommunity(Community, PeerObserver):
    community_id = os.urandom(20)

    def on_peer_added(self, peer: Peer) -> None:
        print("I am:", self.my_peer, "I found:", peer)

    def on_peer_removed(self, peer: Peer) -> None:
        pass

    def started(self) -> None:
        self.network.add_peer_observer(self)


async def start_communities() -> None:
    for i in [1, 2]:
        builder = ConfigBuilder().clear_keys().clear_overlays()
        builder.add_key("my peer", "medium", f"ec{i}.pem")
        # We provide the 'started' function to the 'on_start'.
        # We will call the overlay's 'started' function without any
        # arguments once IPv8 is initialized.
        builder.add_overlay("MyCommunity", "my peer",
                            [WalkerDefinition(Strategy.RandomWalk,
                                              10, {'timeout': 3.0})],
                            default_bootstrap_defs, {}, [('started',)])
        await IPv8(builder.finalize(),
                   extra_communities={'MyCommunity': MyCommunity}).start()
    await run_forever()
```

(continues on next page)

(continued from previous page)

```
run(start_communities())
```

Running this should yield something like the following output:

```
$ python main.py
I am: Peer<0.0.0.0:0:8090, dxGFpQ4awTMz826HOVCB5OoiPPI=> I found: Peer<0.0.0.0:0:8091,␣
↪YfHrKJR4O72/k/FBYYxMIQwOb1U=>
I am: Peer<0.0.0.0:0:8091, YfHrKJR4O72/k/FBYYxMIQwOb1U=> I found: Peer<0.0.0.0:0:8090,␣
↪dxGFpQ4awTMz826HOVCB5OoiPPI=>
```

Warning: You should never use the `address` of a `Peer` as its identifier. A `Peer`'s `address` can change over time! Instead, use the `mid` of a Peer (which is the `SHA-1` hash of its public key) or its `public_key.key_to_bin()` (the serialized form of the public key). The public key of a `Peer` never changes.

6.2.5 Adding messages

As an example for adding messages, we will now make a Lamport clock for three peers. Update your `main.py` once again to contain the following code:

```
import os
from asyncio import run
from dataclasses import dataclass

from ipv8.community import Community, CommunitySettings
from ipv8.configuration import ConfigBuilder, Strategy, WalkerDefinition, default_
↪bootstrap_defs
from ipv8.lazy_community import lazy_wrapper
from ipv8.messaging.payload_dataclass import overwrite_dataclass
from ipv8.types import Peer
from ipv8.util import run_forever
from ipv8_service import IPv8

# Enhance normal dataclasses for IPv8 (see the serialization documentation)
dataclass = overwrite_dataclass(dataclass)


@dataclass(msg_id=1)  # The value 1 identifies this message and must be unique per␣
↪community
class MyMessage:
    clock: int  # We add an integer (technically a "long long") field "clock" to this␣
↪message


class MyCommunity(Community):
    community_id = os.urandom(20)

    def __init__(self, settings: CommunitySettings) -> None:
        super().__init__(settings)
        # Register the message handler for messages (with the identifier "1").
```

(continues on next page)

(continued from previous page)

```
        self.add_message_handler(MyMessage, self.on_message)
        # The Lamport clock this peer maintains.
        # This is for the example of global clock synchronization.
        self.lamport_clock = 0

    def started(self) -> None:
        async def start_communication() -> None:
            if not self.lamport_clock:
                # If we have not started counting, try boostrapping
                # communication with our other known peers.
                for p in self.get_peers():
                    self.ez_send(p, MyMessage(self.lamport_clock))
            else:
                self.cancel_pending_task("start_communication")

        # We register an asyncio task with this overlay.
        # This makes sure that the task ends when this overlay is unloaded.
        # We call the 'start_communication' function every 5.0 seconds, starting now.
        self.register_task("start_communication", start_communication, interval=5.0,␣
↪delay=0)

    @lazy_wrapper(MyMessage)
    def on_message(self, peer: Peer, payload: MyMessage) -> None:
        # Update our Lamport clock.
        self.lamport_clock = max(self.lamport_clock, payload.clock) + 1
        print(self.my_peer, "current clock:", self.lamport_clock)
        # Then synchronize with the rest of the network again.
        self.ez_send(peer, MyMessage(self.lamport_clock))


async def start_communities() -> None:
    for i in [1, 2, 3]:
        builder = ConfigBuilder().clear_keys().clear_overlays()
        builder.add_key("my peer", "medium", f"ec{i}.pem")
        builder.add_overlay("MyCommunity", "my peer",
                            [WalkerDefinition(Strategy.RandomWalk,
                                              10, {'timeout': 3.0})],
                            default_bootstrap_defs, {}, [('started',)])
        await IPv8(builder.finalize(),
                   extra_communities={'MyCommunity': MyCommunity}).start()
    await run_forever()


run(start_communities())
```

If you run this, you should see the three peers actively trying to establish an ever-increasing global clock value.

6.3 Storing states in IPv8

This document assumes you have a basic understanding of network overlays in IPv8, as documented in the overlay tutorial. You will learn how to use the IPv8's `RequestCache` class to store the state of message flows.

6.3.1 When you need a state

More often than not messages come in *flows*. For example, one peer sends out a *request* and another peer provides a *response*. Or, as another example, your message is too big to fit into a single UDP packet and you need to keep track of multiple smaller messages that belong together. In these cases you need to keep a state. The `RequestCache` class keeps track of states and also natively includes a timeout mechanism to make sure you don't get a memory leak.

Typically, you will use one `RequestCache` per network overlay, to which you add the caches that store states.

6.3.2 The hard way

The most straightforward way of interacting with the `RequestCache` is by adding `NumberCache` instances to it directly. Normally, you will use `add()` and `pop()` to respectively add new caches and remove existing caches from the `RequestCache`. This is a bare-bones example of how states can be stored and retrieved:

```python
from asyncio import create_task, run, sleep

from ipv8.requestcache import NumberCacheWithName, RequestCache


class MyState(NumberCacheWithName):

    name = "my-state"

    def __init__(self, request_cache: RequestCache,
                 identifier: int, state: int) -> None:
        super().__init__(request_cache, self.name, identifier)
        self.state = state


async def foo(request_cache: RequestCache) -> None:
    """
    Add a new MyState cache to the global request cache.
    The state variable is set to 42 and the identifier of this cache is 0.
    """
    cache = MyState(request_cache, 0, 42)
    request_cache.add(cache)


async def bar() -> None:
    """
    Wait until a MyState cache with identifier 0 is added.
    Then, remove this cache from the global request cache and print its state.
    """
    # Normally, you would add this to a network overlay instance.
    request_cache = RequestCache()
```

(continues on next page)

(continued from previous page)

```
    _ = create_task(foo(request_cache))

    while not request_cache.has(MyState, 0):
        await sleep(0.1)
    cache = request_cache.pop(MyState, 0)
    print("I found a cache with the state:", cache.state)


run(bar())
```

In the previous example we have assumed that a cache would eventually arrive. This will almost never be the case in practice. You can overwrite the `on_timeout` method of your `NumberCache` instances to deal with cleaning up when a cache times out. In this following example we shut down when the cache times out:

```
from asyncio import run, sleep

from ipv8.requestcache import NumberCacheWithName, RequestCache


class MyState(NumberCacheWithName):

    name = "my-state"

    def __init__(self, request_cache: RequestCache,
                 identifier: int, state: int) -> None:
        super().__init__(request_cache, self.name, identifier)
        self.state = state

    def on_timeout(self) -> None:
        print("Oh no! I never received a response!")

    @property
    def timeout_delay(self) -> float:
        # We will timeout after 3 seconds (default is 10 seconds)
        return 3.0


async def foo() -> None:
    request_cache = RequestCache()
    cache = MyState(request_cache, 0, 42)
    request_cache.add(cache)
    await sleep(4)


run(foo())
```

You may notice some inconvenient properties of these caches. You need to generate a unique identifier and manually keep track of it. This is why we have an easier way to interact with the `RequestCache`.

6.3.3 The easier way

Let's look at the complete Community code for two peers that use each other to count to 10. For this toy box example we define two messages and a single cache. Unlike when doing things the hard way, we now use a `RandomNumberCache` to have IPv8 select a message identifier for us. Both the `identifier` fields for the messages and the `name` for the cache are required. Please attentively read through this code:

```python
# We'll use this global variable to keep track of the IPv8 instances that finished.
DONE = []


@vp_compile
class MyRequest(VariablePayload):
    msg_id = 1
    format_list = ['I', 'I']
    names = ["value", "identifier"]


@vp_compile
class MyResponse(VariablePayload):
    msg_id = 2
    format_list = ['I', 'I']
    names = ["value", "identifier"]


class MyCache(RandomNumberCacheWithName):
    name = "my-cache"

    def __init__(self, request_cache: RequestCache, value: int) -> None:
        super().__init__(request_cache, self.name)
        self.value = value


class MyCommunity(Community):
    community_id = os.urandom(20)

    def __init__(self, settings: CommunitySettings) -> None:
        super().__init__(settings)
        self.add_message_handler(1, self.on_request)
        self.add_message_handler(2, self.on_response)

        # This is where the magic starts: add a ``request_cache`` variable.
        self.request_cache = RequestCache()

    async def unload(self) -> None:
        # Don't forget to shut down the RequestCache when you unload the Community!
        await self.request_cache.shutdown()
        await super().unload()

    def started(self) -> None:
        self.register_task("wait for peers and send a request", self.send)

    async def send(self) -> None:
```

(continues on next page)

(continued from previous page)

```
        # Wait for our local peers to connect to eachother.
        while not self.get_peers():
            await sleep(0.1)
        # Then, create and register our cache.
        cache = self.request_cache.add(MyCache(self.request_cache, 0))
        # If the overlay is shutting down the cache will be None.
        if cache is not None:
            # Finally, send the overlay message to the other peer.
            for peer in self.get_peers():
                self.ez_send(peer, MyRequest(cache.value, cache.number))

    @lazy_wrapper(MyRequest)
    def on_request(self, peer: Peer, payload: MyRequest) -> None:
        # Our service is to increment the value of the request and send this in the␣
↪response.
        self.ez_send(peer, MyResponse(payload.value + 1, payload.identifier))

    @lazy_wrapper(MyResponse)
    @retrieve_cache(MyCache)
    def on_response(self, peer: Peer, payload: MyResponse, cache: MyCache) -> None:
        print(peer, "responded to:", cache.value, "with:", payload.value)

        # Stop the experiment if both peers reach a value of 10.
        if payload.value == 10:
            DONE.append(True)
            return

        # Otherwise, do the same thing over again and ask for another increment.
        cache = self.request_cache.add(MyCache(self.request_cache, payload.value))
        if cache is not None:
            for peer in self.get_peers():
                self.ez_send(peer, MyRequest(payload.value, cache.number))
                # To spice things up, we'll perform a replay attack.
                # The RequestCache causes this second duplicate message to be ignored.
                self.ez_send(peer, MyRequest(payload.value, cache.number))
```

You are encouraged to play around with this code. Also, take notice of the fact that this example includes a replay attack (try removing the cache and see what happens).

6.4 Unit Testing Overlays

This document assumes you have a basic understanding of network overlays in IPv8, as documented in the overlay tutorial. You will learn how to use the IPv8's `TestBase` class to unit test your overlays.

6.4.1 Files

This tutorial will place all of its files in the `~/Documents/ipv8_tutorial` directory. You are free to choose whatever directory you want, to place your files in. This tutorial uses the following files in the working directory:

```
community.py
test_community.py
```

We will use the following `community.py` in this tutorial:

```python
import os
import unittest
from typing import TYPE_CHECKING, Any

from ipv8.community import Community, CommunitySettings
from ipv8.requestcache import NumberCache, RequestCache
from ipv8.test.base import TestBase
from ipv8.test.mocking.ipv8 import MockIPv8

if TYPE_CHECKING:
    from ipv8.messaging.payload import IntroductionRequestPayload
    from ipv8.messaging.payload_headers import GlobalTimeDistributionPayload
    from ipv8.types import Peer


class MyCache(NumberCache):

    def __init__(self, request_cache: RequestCache, overlay: MyCommunity) -> None:
        super().__init__(request_cache, "", 0)
        self.overlay = overlay

    def on_timeout(self) -> None:
        self.overlay.timed_out = True


class MyCommunitySettings(CommunitySettings):
    some_constant: int | None = None


class MyCommunity(Community):
    community_id = os.urandom(20)
    settings_class = MyCommunitySettings

    def __init__(self, settings: MyCommunitySettings) -> None:
        super().__init__(settings)
        self.request_cache = RequestCache()

        self._some_constant = 42 if settings.some_constant is None else settings.some_
↪constant
        self.last_peer = None
        self.timed_out = False

    async def unload(self) -> None:
        await self.request_cache.shutdown()
```

(continues on next page)

(continued from previous page)

```
        await super().unload()

    def some_constant(self) -> int:
        return self._some_constant

    def introduction_request_callback(self, peer: Peer,
                                      dist: GlobalTimeDistributionPayload,
                                      payload: IntroductionRequestPayload) -> None:
        self.last_peer = peer

    def add_cache(self) -> None:
        self.request_cache.add(MyCache(self.request_cache, self))
```

You're encouraged to fill `test_community.py` yourself as you read through this tutorial.

6.4.2 Why and How?

After playing around with your first overlay, you may have discovered that running multiple processes and configuring your communities to test functionality is not very easy or reproducible. We certainly have. Therefore, we have created the `TestBase` class with all the tools you need to mock the Internet and make beautiful unit tests.

Because `TestBase` is a subclass of `unittest.TestCase` you can use common unit testing convenience methods, like `testEqual`, `testTrue`, `setUp`, `setUpClass`, etc. This also means that `TestBase` can be used with just about any test runner out there (like `unittest`, `nosetests` or `pytest`).

The way we will run our unit tests in this tutorial is with:

```
python3 -m unittest test_community.py
```

If you have custom logic in your subclass, please make sure to call your `super()` methods. Here's an example of custom `setUp` and `tearDown` methods:

```
class MyTests(TestBase[MyCommunity]):

    def setUp(self) -> None:
        super().setUp()
        # Insert your setUp logic here

    async def tearDown(self) -> None:
        await super().tearDown()
        # Insert your tearDown logic here
```

6.4.3 Deadlock Detection

Before you start testing, you need to be warned about `TestBase.MAX_TEST_TIME`. By default, `TestBase.MAX_TEST_TIME` is set to 10 seconds. This means that if your testing class takes more than 10 seconds, `TestBase` will terminate it.

We should probably mention that in proper software engineering a unit test case should never take 10 seconds. However, we're not here to judge. If you want this timeout increased, simply overwrite the value of `MAX_TEST_TIME` in your subclass. For example:

```
class MyTests(TestBase):

    MAX_TEST_TIME = 30.0  # Now this class can take 30 seconds
```

6.4.4 Creating Instances

The `initialize()` method takes care of initializing your `Community` subclass for you. It's as easy as this:

```
    async def test_call(self) -> None:
        """
        Create a MyCommunity and check the output of some_constant().
        """
        # Create 1 MyCommunity
        self.initialize(MyCommunity, 1)

        # Nodes are 0-indexed
        value = self.overlay(0).some_constant()

        self.assertEqual(42, value)
```

What happened here? First, we instructed `TestBase` to create 1 instance of `MyCommunity` using `initialize()`. As a side note: the raw information needed to make this happen (the mocking of the Internet and the interconnection of overlays) is actually stored in the `nodes` list of `TestBase`. Second, we ask our `TestBase` to give us the overlay instance of node 0, which is our only node. The `overlay()` method is one of the many convenience methods in `TestBase` to access common data in overlays. We'll provide a complete list of these convenience methods later in this document. Last, we use a common `unittest.TestCase` assertion to check if our `some_constant()` overlay method returned 42.

In some cases, you might need to give additional parameters to your `Community`'s `__init__()` method. In these cases, you can simply add additional keyword arguments to `initialize()`.

```
    async def test_call2(self) -> None:
        """
        Create a MyCommunity with a custom some_constant.
        """
        self.initialize(MyCommunity, 1, MyCommunitySettings(some_constant=7))

        value = self.overlay(0).some_constant()

        self.assertEqual(7, value)
```

In yet more advanced use cases, you may want to provide your own `MockIPv8` instances. This will usually be the case if your `Community` instance only supports specific keys. Commonly, `Community` instances may choose to **only** support `curve25519` keys, which you can do as follows:

```
    def create_node(self, *args: Any, **kwargs) -> MockIPv8:
        return MockIPv8("curve25519", self.overlay_class, *args, **kwargs)
```

6.4.5 Communication

You should now be able to create `Community` instances and call their methods. However, these instances are not communicating with each other yet. Take note of this code in our `Community` instance that stores the last peer that sent us an introduction request:

```
    def introduction_request_callback(self, peer: Peer,
                                      dist: GlobalTimeDistributionPayload,
                                      payload: IntroductionRequestPayload) -> None:
        self.last_peer = peer
```

This code simply stores whatever `Peer` object last sent us a request. We'll create a unit test to test whether this happened:

```
    async def test_intro_called(self) -> None:
        """
        Check if we got a request from another MyCommunity.
        """
        self.initialize(MyCommunity, 2)

        # We have the overlay of Peer 0 send a message to Peer 1.
        self.overlay(0).send_introduction_request(self.peer(1))
        # Our test is running in the asyncio main thread!
        # Let's yield to allow messages to be passed.
        await self.deliver_messages()

        # Peer 1 should have received the message from Peer 0.
        self.assertEqual(self.peer(0), self.overlay(1).last_peer)
```

Let's run through this example. First we create two instances of `MyCommunity` using `initialize()`. Second, we instruct the first node in our test to send a message to our second node. Here `send_introduction_request()` creates and sends a message to another peer and `deliver_messages()` allows it to be received. Lastly, we assert that our second node received a message from our first node.

Note that the `asyncio` programming model of Python executes its events on the main thread (the event loop), including this test case and the communication that is caused by it. In other words, since the test itself is occupying the main thread, the messaging will only happen after our test is finished! By the time it is allowed to execute, the communication is already cancelled. The `deliver_messages()` method backs off for a given amount of time and then waits for the main thread to be freed.

Now comes the caveat. The main thread being freed may not mean your `Community` is actually done doing stuff. **It is possible to schedule asyncio events in such a way that deliver_messages() can't detect them.** This commonly happens when you use threading or hardware (like sockets). In these exceptional cases, you can use `asyncio.sleep()` or, better yet, await a custom `Future` in the test.

6.4.6 Piggybacking on Introductions

Some `Community` instances prefer to piggyback information onto introductions. As `TestBase` simply adds peers to each other directly, this piggybacked information is not sent. The `introduce_nodes()` method allows you to send these introductions anyway, used as follows (note the absence of `deliver_messages()`):

```
    async def test_intro_called2(self) -> None:
        """
        Check if we got a request from another MyCommunity.
        """
```

(continues on next page)

(continued from previous page)

```
        self.initialize(MyCommunity, 2)

        await self.introduce_nodes()

        self.assertEqual(self.peer(0), self.overlay(1).last_peer)
        self.assertEqual(self.peer(1), self.overlay(0).last_peer)
```

6.4.7 Using the RequestCache

In real `Community` instances, you will have many timeouts and lots of timeout logic in caches. To make it easier to trigger these timeouts in the `RequestCache`, we use the `passthrough()` method. Here's an example:

```
    async def test_passthrough(self) -> None:
        """
        Check if a cache time out is properly handled.
        """
        self.initialize(MyCommunity, 1)

        with self.overlay(0).request_cache.passthrough():
            self.overlay(0).add_cache()
        await self.deliver_messages()

        self.assertTrue(self.overlay(0).timed_out)
```

In this example we use the `passthrough()` contextmanager while we invoke a function that adds a cache. This causes the timeout of the `MyCache` cache we add inside `add_cache` to be nullified and instantly fire. Do note that this timeout occurs in the `asyncio` event loop and we need to allow it to fire. To yield the main thread we use `deliver_messages()` again (though in this case `await asyncio.sleep(0.0)` would have also done the trick).

In some complex cases you may have more than one type of cache being added. In these cases you can add a filter to `passthrough()` to make it only nullify some particular classes (simply add these classes as arguments to `passthrough()`):

```
    async def test_passthrough2(self) -> None:
        """
        Check if a cache time out is properly handled.
        """
        self.initialize(MyCommunity, 1)

        with self.overlay(0).request_cache.passthrough(MyCache):
            self.overlay(0).add_cache()
        await self.deliver_messages()

        self.assertTrue(self.overlay(0).timed_out)
```

6.4.8 Fragile Packet Handling

By default IPv8 adds a general exception handler in Community instances, to disallow external messages crashing you. However, when testing, this exception handler is removed by TestBase. If you want to enable the general exception handler again, you can either add your class to the production_overlay_classes list or overwrite TestBase.patch_overlays(). For example:

```
def patch_overlays(self, i: int) -> None:
    if i == 1:
        pass  # We'll run the general exception handler for Peer 1
    else:
        super().patch_overlays(i)
```

6.4.9 Temporary Files

In some cases, you may require temporary files in your unit tests. TestBase exposes the temporary_directory() method to generate directories for these files. This method is random.seed() compatible. Normally, TestBase will clean up these files automatically for you. However, if you hard-crash TestBase before its tearDown is invoked, the temporary directories will not be cleaned up.

The temporary directory names are prefixed with _temp_ and use a uuid as a unique name. The temporary directories will be created in the current working directory for the mechanism to work on all supported platforms (Windows, Mac and Linux) even with limited permissions.

6.4.10 Asserting Message Delivery

You may want to assert that an overlay receives certain messages after a function. The TestBase exposes the assertReceivedBy() function to do just that. We'll run through its functionality by example.

Most of the time, you will want to check if a peer received certain messages. In the following example peer 0 first sends message 1 and then sends message 2 to peer 1. The following construction asserts this:

```
with self.assertReceivedBy(1, [Message1, Message2]):
    self.overlay(0).send_msg_to(self.peer(1), 1)
    self.overlay(0).send_msg_to(self.peer(1), 2)
    await self.deliver_messages()
```

Sometimes, you can't be sure in what order messages are sent. In these cases you can use ordered=False:

```
with self.assertReceivedBy(1, [Message1, Message2, Message2], ordered=False):
    messages = [2, 1, 2]
    shuffle(messages)
    self.overlay(0).send_msg_to(self.peer(1), messages[0])
    self.overlay(0).send_msg_to(self.peer(1), messages[1])
    self.overlay(0).send_msg_to(self.peer(1), messages[2])
    await self.deliver_messages()
```

In other cases, your overlay may be sending messages which you cannot control and/or which you don't care about. In these cases you can set a filter to only include the messages you want:

```
with self.assertReceivedBy(1, [Message1, Message2], message_filter=[Message1,␣
↪Message2]):
    self.overlay(0).send_msg_to(self.peer(1), 1)
```

(continues on next page)

(continued from previous page)

```
        if random() > 0.5:
            self.overlay(0).send_msg_to(self.peer(1), 3)
        self.overlay(0).send_msg_to(self.peer(1), 2)
        await self.deliver_messages()
```

It may also be helpful to inspect the contents of each payload. You can simply use the return value of the assert function to perform further inspection:

```
    with self.assertReceivedBy(1, [Message1, Message2]) as received_messages:
        self.overlay(0).send_msg_to(self.peer(1), 1)
        self.overlay(0).send_msg_to(self.peer(1), 2)
        await self.deliver_messages()

    message1, message2 = received_messages
    self.assertEqual(1, message1.value)
    self.assertEqual(2, message2.value)
```

If you want to use `assertReceivedBy()`, make sure that:

1. Your overlay message handlers only handle a single payload.
2. Your messages specify a `msg_id`.
3. Your messages are compatible with `ez_send()` and `lazy_wrapper_*()`.

6.4.11 Shortcut Reference

As usual, there is an easy and a hard way to do everything in IPv8. You are welcome to call `self.nodes[i].my_peer.public_key.key_to_bin()` manually every time you wish to access the public key of node `i`. Or, instead, you may use the available shortcut `self.key_bin(i)`. You may find your unit test become a lot more readable if you use the available `TestBase` shortcuts though.

Table 1: Available `TestBase` shortcuts

method	description
address(i)	The IPv4 address of peer i.
endpoint(i)	The Endpoint instance of peer i.
key_bin(i)	The serialized public key (bytes) of peer i.
key_bin_private(i)	The serialized private key (bytes) of peer i.
mid(i)	The SHA-1 of the public key of peer i.
my_peer(i)	The private my_peer Peer instance of peer i.
network(i)	The Network instance of peer i.
node(i)	The MockIPv8 instance of peer i.
overlay(i)	The Community instance of peer i.
peer(i)	The public Peer instance of peer i.
private_key(i)	The private key instance of peer i.
public_key(i)	The public key instance of peer i.

You are encouraged to add shortcuts that may be relevant to your own `Community` instance in your own test class.

6.5 Task Management

This document assumes you have a basic understanding of network overlays in IPv8, as documented in the overlay tutorial. You will learn how to use the IPv8's `TaskManager` class to manage `asyncio` tasks and how to use `NumberCache` to manage `Future` instances.

6.5.1 What is a task?

Essentially, a task is a way for `asyncio` to point to code that should be executed at some point. The `asyncio` library checks if it has something to execute and executes it until it has no more tasks to execute. However, there are many intricacies when dealing with tasks. Consider the following example:

```python
import asyncio

COMPLETED = []


async def execute_me_too(i: int) -> None:
    await asyncio.sleep(0.5)
    COMPLETED.append(i)


async def execute_me() -> None:
    execute_me_too(1)  # 1
    await execute_me_too(2)  # 2
    COMPLETED.append(3)  # 3
    _ = asyncio.ensure_future(execute_me_too(4))  # 4
    COMPLETED.append(5)  # 5
    await asyncio.sleep(1)  # 6

asyncio.run(execute_me())
print(COMPLETED)
```

Can you guess what will be printed?

The correct answer is `[2, 3, 5, 4]` and equally important is that the answer changes to `[2, 3, 5]` if you omit the final `await asyncio.sleep(1)`. Feel free to skip to the "The TaskManager" section if both of these answers are obvious to you.

Let's run through `execute_me()` to explain this behavior:

1. The `execute_me_too(1)` will not be executed, though your application will not crash. You will be informed of this through the `RuntimeWarning: coroutine 'execute_me_too' was never awaited` warning message. You should have awaited this `execute_me_too` call, we do this properly in the next line.
2. By awaiting `execute_me_too(2)` the `execute_me()` call will wait until `execute_me_too(2)` has finished executing. This adds the value `2` to the `COMPLETED` list.
3. After waiting for `execute_me_too(2)` to finish `COMPLETED.append(3)` is allowed to append the value `3` to the `COMPLETED` list.
4. By creating a future for `execute_me_too(4)`, we allow `execute_me()` to continue executing.
5. While we wait for `execute_me_too(4)` to finish, `COMPLETED.append(5)` can already access the `COMPLETED` list and insert its value `5`.
6. While `execute_me()` waits for another second, `execute_me_too(4)` is allowed to add to the `COMPLETED` list.

Note that if you had omitted `await asyncio.sleep(1)`, the `execute_me()` call would not be waiting for anything anymore and return (outputting `[2, 3, 5]`). Instead, the future returned by `asyncio.ensure_future()` should have been awaited, as follows:

```python
async def execute_me() -> None:
    await execute_me_too(2)
    COMPLETED.append(3)
    fut = asyncio.ensure_future(execute_me_too(4))  # store future
    COMPLETED.append(5)
    await fut  # await future before exiting
```

As an added bonus, this is also faster than the previous method. This new example will finish when the `execute_me_too(4)` call completes, instead of after waiting a full second.

This concludes the basics of `asyncio` tasks. Now, we'll show you how to make your life easier by using IPv8's `TaskManager` class.

6.5.2 The TaskManager

Managing `Future` instances and `coroutine` instances is complex and error-prone and, to help you with managing these, IPv8 has the `TaskManager` class. The `TaskManager` takes care of managing all of your calls that have not completed yet and allows you to cancel them on demand. You can even find this class in a separate PyPi repository: https://pypi.org/project/ipv8-taskmanager/

Adding tasks

To add tasks to your `TaskManager` you can call `register_task()`. You register a task with a name, which you can later use to inspect the state of a task or cancel it. For example:

```python
import asyncio

from ipv8.taskmanager import TaskManager

COMPLETED = []


async def execute_me_too(i: int) -> None:
    await asyncio.sleep(0.5)
    COMPLETED.append(i)


async def main() -> None:
    task_manager = TaskManager()

    task_manager.register_task("execute_me_too1", execute_me_too, 1)
    task_manager.register_task("execute_me_too2", execute_me_too, 2)
    task_manager.cancel_pending_task("execute_me_too1")
    await task_manager.wait_for_tasks()

    await task_manager.shutdown_task_manager()
    print(COMPLETED)

asyncio.run(main())
```

This example prints [2]. If you had not called wait_for_tasks() before shutdown_task_manager() in this example, all of your registered tasks would have been canceled, printing [].

In some cases you may also not be interested in canceling a particular task. For this use case the register_anonymous_task() call exists, for example:

```
    for i in range(20):
        task_manager.register_anonymous_task("execute_me_too",
                                             execute_me_too, i)
    await task_manager.wait_for_tasks()
```

Note that this example takes just over half a second to execute, all 20 calls to execute_me_too() are scheduled at (almost) the same time!

Periodic and delayed tasks

Next to simply adding tasks, the TaskManager also allows you to invoke calls after a delay or periodically. The following example will add the value 1 to the COMPLETED list periodically and inject a single value of 2 after half a second:

```
async def execute_me_too(i: int, task_manager: TaskManager) -> None:
    if len(COMPLETED) == 20:
        task_manager.cancel_pending_task("keep adding 1")
        return
    COMPLETED.append(i)


async def main() -> None:
    task_manager = TaskManager()

    task_manager.register_task("keep adding 1", execute_me_too,
                               1, task_manager, interval=0.1)
    task_manager.register_task("sneaky inject", execute_me_too,
                               2, task_manager, delay=0.5)
    await task_manager.wait_for_tasks()

    await task_manager.shutdown_task_manager()
    print(COMPLETED)
```

This example prints [1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1].

Note that you can also create a task with both an interval and a delay.

Community Integration

For your convenience, every Community is also a TaskManager. However, try to avoid spawning any tasks in the __init__ of your Community. Specifically, you could end up scheduling a task in between your Community and other Community initialization. Working with a half-initialized IPv8 may lead to problems. This practice also makes your Community more difficult to test.

If you do end up in a situation where you absolutely must schedule a task from the __init__(), you can use the TaskManager's cancel_all_pending_tasks() method to cancel all registered tasks. This is an example of how to deal with this in your unit tests:

```python
class TestMyCommunity(TestBase):

    def create_node(self, *args: Any, **kwargs) -> MockIPv8:
        mock_ipv8 = super().create_node(*args, **kwargs)
        mock_ipv8.overlay.cancel_all_pending_tasks()
        return mock_ipv8

    async def test_something(self) -> None:
        self.initialize(MyCommunity, 1)  # Will not run tasks
```

Futures and caches

Sometimes you may find that certain tasks belong to a message context. In other words, you may have a task that belongs to a *cache* (see the storing states tutorial). By registering a `Future` instance in your `NumberCache` subclass it will automatically be canceled when the `NumberCache` gets canceled or times out. You can do so using the `register_future()` method. This is a complete example:

```python
import asyncio

from ipv8.requestcache import NumberCacheWithName, RequestCache


class MyCache(NumberCacheWithName):

    name = "my cache"

    def __init__(self,
                 request_cache: RequestCache,
                 number: int) -> None:
        super().__init__(request_cache, self.name, number)

        self.awaitable = asyncio.Future()

        self.register_future(self.awaitable, on_timeout=False)

    @property
    def timeout_delay(self) -> float:
        return 1.0

    def finish(self) -> None:
        self.awaitable.set_result(True)


async def main() -> None:
    rq = RequestCache()

    rq.add(MyCache(rq, 0))
    with rq.passthrough():
        rq.add(MyCache(rq, 1))  # Overwritten timeout = 0.0
    rq.add(MyCache(rq, 2))

    future0 = rq.get(MyCache, 0).awaitable
```

(continues on next page)

(continued from previous page)

```
    future1 = rq.get(MyCache, 1).awaitable
    future2 = rq.get(MyCache, 2).awaitable

    rq.get(MyCache, 0).finish()
    await future0
    print(f"future0.result()={future0.result()}")
    rq.pop(MyCache, 0)

    await future1
    print(f"future1.result()={future1.result()}")

    await rq.shutdown()

    print(f"future2.cancelled()={future2.cancelled()}")


asyncio.run(main())
```

This example prints:

```
future0.result()=True
future1.result()=False
future2.cancelled()=True
```

This example showcases the three states your `Future` may find itself in. First, it may have been called and received an explicit result (`True` in this case). Second, the future may have timed out. By default a timed-out `Future` will have its result set to `None`, but you can even give this method an exception class to have it raise an exception for you. In this example we set the value to `False`. Third and last is the case where your future was cancelled, which will raise a `asyncio.exceptions.CancelledError` if you `await` it.

6.6 Network IO and the DiscoveryStrategy

This document assumes you have a basic understanding of asyncio tasks, as documented in the tasks tutorial. You will learn how to use the IPv8's `DiscoveryStrategy` class to avoid network congestion.

6.6.1 The DiscoveryStrategy

IPv8 only manages one socket (`Endpoint`), which is most likely using the UDP protocol. If every `Community` starts sending at the exact same time and overpowers the UDP socket, this causes packet drops. To counter this, IPv8 has the `DiscoveryStrategy` class.

An IPv8 instance will call each of its registered `DiscoveryStrategy` instances sequentially to avoid network I/O clashes. If you have an `interval` task in your `TaskManager` that leads to network I/O, you should consider converting it to a `DiscoveryStrategy`. You can make your own subclass as follows:

```
class MyDiscoveryStrategy(DiscoveryStrategy):

    def take_step(self) -> None:
        with self.walk_lock:
            # Insert your logic here. For example:
```

(continues on next page)

(continued from previous page)

```
            if self.overlay.get_peers():
                peer = choice(self.overlay.get_peers())
                self.overlay.send_introduction_request(peer)
```

Note that a `DiscoveryStrategy` should be thread-safe. You can use the `walk_lock` for thread safety.

6.6.2 Using a DiscoveryStrategy

You can register your `DiscoveryStrategy` with a running `IPv8` instance as follows:

```
def main(ipv8_instance: IPv8) -> None:
    overlay = ipv8_instance.get_overlay(MyCommunity)
    target_peers = -1
    ipv8_instance.add_strategy(overlay,
                               MyDiscoveryStrategy(overlay),
                               target_peers)
```

Note that we specify a `target_peers` argument. This argument specifies the amount of peers after which the `DiscoveryStrategy` should no longer be called. Calls will be resumed when the amount of peers in your `Community` dips below this value again. For example, the built-in `RandomWalk` strategy can be configured to stop finding new peers after if an overlay already has `20` or more peers. In this example we have used the magic value `-1`, which causes `IPv8` to never stop calling this strategy.

You can also load your strategy through the `configuration` or `loader`. First, an example of how to do this with the `configuration`:

```
class MyCommunity(Community):
    community_id = os.urandom(20)

    def get_available_strategies(self) -> dict[str, type[DiscoveryStrategy]]:
        return {"MyDiscoveryStrategy": MyDiscoveryStrategy}


definition = {
    'strategy': "MyDiscoveryStrategy",
    'peers': -1,
    'init': {}
}

config = get_default_configuration()
config['overlays'] = [{
    'class': 'MyCommunity',
    'key': "anonymous id",
    'walkers': [definition],
    'bootstrappers': [DISPERSY_BOOTSTRAPPER.copy()],
    'initialize': {},
    'on_start': []
}]
```

Note that you can add as many strategies as you want to an overlay. Also note that for IPv8 to link the name `"MyDiscoveryStrategy"` to a class, you need to define it in your `Community`'s `get_available_strategies()` dictionary.

Lastly, alternatively, the way to add your custom MyDiscoveryStrategy class to a CommunityLauncher is as follows:

```
@overlay('my_module.some_submodule', 'MyCommunity')
@walk_strategy(MyDiscoveryStrategy)
class MyLauncher(CommunityLauncher):
    pass
```

This is the shortest way.

6.7 Using the IPv8 attestation service

This document assumes you have a basic understanding of network overlays in IPv8, as documented in the overlay tutorial. You will learn how to use the IPv8 attestation *HTTP REST API*. This tutorial will use curl to perform HTTP GET and POST requests.

This document will cover the basic flows of identification. If you plan on using real identity data, you will need to familiarize yourself with the the advanced identity controls.

6.7.1 Running the IPv8 service

Fill your main.py file with the following code (runnable with python3 main.py):

```
from asyncio import run
from base64 import b64encode

from ipv8.configuration import get_default_configuration
from ipv8.REST.rest_manager import RESTManager
from ipv8.util import run_forever
from ipv8_service import IPv8


async def start_community() -> None:
    for peer_id in [1, 2]:
        configuration = get_default_configuration()
        configuration['keys'] = [
            {'alias': "anonymous id", 'generation': "curve25519", 'file': f"keyfile_
↪{peer_id}.pem"}]
        configuration['working_directory'] = f"state_{peer_id}"
        configuration['overlays'] = []

        # Start the IPv8 service
        ipv8 = IPv8(configuration)
        await ipv8.start()
        rest_manager = RESTManager(ipv8)
        await rest_manager.start(14410 + peer_id)

        # Print the peer for reference
        print("Starting peer", b64encode(ipv8.keys["anonymous id"].mid))

    await run_forever()
```

(continues on next page)

(continued from previous page)

```
run(start_community())
```

Running the service should yield something like the following output in your terminal:

```
$ python3 main.py
Starting peer aQVwz9aRMRypGwBkaxGRSdQs80c=
Starting peer bPyWPyswqXMhbW8+0RS6xUtNJrs=
```

You should see two messages with 28 character base64 encoded strings. These are the identifiers of the two peers we launched using the service. You can use these identifiers for your reference when playing around with sending attestation requests. In your experiment you will create unique keys and therefore see other identifiers than the `aQVwz9aRMRypGwBkaxGRSdQs80c=` and `bPyWPyswqXMhbW8+0RS6xUtNJrs=` shown above.

6.7.2 Functionality flows

Generally speaking there are two (happy) flows when using the IPv8 attestation framework. The first flow is the enrollment of an attribute and the second flow is the verification of an existing/enrolled attribute. Both flows consist of a distinct set of requests (and responses) which we will explain in detail in the remainder of this document.

To test a flow, we start the two peers we created previously. If you did not remove the key files (`*.pem`) after the first run, you will start the same two peers as in the last run. In our case the output of starting the service is as follows:

```
$ python main.py
Starting peer aQVwz9aRMRypGwBkaxGRSdQs80c=
Starting peer bPyWPyswqXMhbW8+0RS6xUtNJrs=
```

In our case this means that peer `aQVwz9aRMRypGwBkaxGRSdQs80c=` exposes its REST API at `http://localhost:14411/` and peer `bPyWPyswqXMhbW8+0RS6xUtNJrs=` exposes its REST API at `http://localhost:14412/`. If you did not modify the ports in the initial scripts, you will have two different peer identifiers listening at the same ports. For convenience we will refer to our first peer as *Peer 1* and our second peer as *Peer 2*.

As a last note, beware of URL encoding: when passing these identifiers they need to be properly formatted (+ and = are illegal characters). In our case we need to use the following formatting of the peer identifiers in URLs (for Peer 1 and Peer 2 respectively):

```
TGliTmFDTFBLOpyBsled71NjFOZfF3L%2Bw0sdAvcM3xI1nM%2Fik6NbRzxmwgFBJRZdQ
↪%2Bh2CURQlwxtFxe33U7oldJtK%2BE1fTk2rOo%3D
TGliTmFDTFBLOg%2Frrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv%2BhSjbssLYmps
↪%2Bjlh9rb250LYD7gEH20%3D
```

If you are using Python, you can make these identifiers URL-safe by calling `urllib.parse.quote(identifier, safe='')`.

Enrollment/Attestation flow

To enroll, or attest, an attribute we will go through the following steps:

1. Sanity checks: Peer 1 and Peer 2 can see each other and have no existing attributes.
2. Peer 1 requests attestation of an attribute by Peer 2.
3. Peer 2 attests to the requested attribute.
4. Peer 1 checks its attributes to confirm successful attestation.

0. SANITY CHECK - First we check if both peers can see each other using their respective interfaces.

```
$ curl http://localhost:14411/identity/pseudonym1/peers
{"peers": ["TGliTmFDTFBLOg/
↪rrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv+hSjbssLYmps+jlh9rb25OLYD7gEH20=
↪"]}
$ curl http://localhost:14412/identity/pseudonym2/peers
{"peers": ["TGliTmFDTFBLOpyBsled71NjFOZfF3L+w0sdAvcM3xI1nM/
↪ik6NbRzxmwgFBJRZdQ+h2CURQlwxtFxe33U7oldJtK+E1fTk2rOo="]}
```

Pseudonyms are lazy-loaded and/or created on demand, it may take a few seconds for the pseudonyms to discover each other. Then we confirm that neither peer has existing attributes.

```
$ curl http://localhost:14411/identity/pseudonym1/credentials
{"names": []}
$ curl http://localhost:14412/identity/pseudonym2/credentials
{"names": []}
```

1. ATTESTATION REQUEST - Peer 1 will now ask Peer 2 to attest to an attribute.

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"name":"my_attribute","schema":
↪"id_metadata","metadata":{}}' "http://localhost:14411/identity/pseudonym1/request/
↪TGliTmFDTFBLOg%2Frrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv%2BhSjbssLYmps
↪%2Bjlh9rb25OLYD7gEH20%3D"
{"success": true}
```

2. ATTESTATION - Peer 2 finds an outstanding request for attestation. Peer 2 will now attest to some attribute value of Peer 1 (dmFsdWU= is the string `value` in base64 encoding).

```
$ curl http://localhost:14412/identity/pseudonym2/outstanding/attestations
{"requests": [{"peer": "TGliTmFDTFBLOpyBsled71NjFOZfF3L+w0sdAvcM3xI1nM/
↪ik6NbRzxmwgFBJRZdQ+h2CURQlwxtFxe33U7oldJtK+E1fTk2rOo=", "attribute_name": "my_attribute
↪", "metadata": "{}"}]}
$ curl -X PUT -H "Content-Type: application/json" -d '{"name":"my_attribute","value":
↪"dmFsdWU="}' "http://localhost:14412/identity/pseudonym2/attest/
↪TGliTmFDTFBLOpyBsled71NjFOZfF3L%2Bw0sdAvcM3xI1nM%2Fik6NbRzxmwgFBJRZdQ
↪%2Bh2CURQlwxtFxe33U7oldJtK%2BE1fTk2rOo%3D"
{"success": true}
```

3. CHECK - Peer 1 confirms that he now has an attested attribute.

```
$ curl http://localhost:14411/identity/pseudonym1/credentials
{"names": [{"name": "my_attribute", "hash": "mtMiZioWORNgV+GeGACsY+rD+lI=", "metadata": {
↪"name": "my_attribute", "schema": "id_metadata", "date": 1593171171.876003}, "attesters
↪": ["TGliTmFDTFBLOg/
```

(continues on next page)

(continued from previous page)

```
↪rrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv+hSjbssLYmps+jlh9rb250LYD7gEH20=
↪"]}]}
$ curl http://localhost:14412/identity/pseudonym2/credentials
{"names": []}
```

Attribute verification flow

To verify an attribute we will go through the following steps:

1. Sanity checks: Peer 1 and Peer 2 can see each other and Peer 1 has an existing attribute.
2. Peer 2 requests verification of an attribute of Peer 1.
3. Peer 1 allows verification of its attribute.
4. Peer 2 checks the verification output for the requested verification.

0. SANITY CHECK - First we check if both peers can see each other using their respective interfaces.

```
$ curl http://localhost:14411/identity/pseudonym1/peers
{"peers": ["TGliTmFDTFBLOg/
↪rrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv+hSjbssLYmps+jlh9rb250LYD7gEH20=
↪"]}
$ curl http://localhost:14412/identity/pseudonym2/peers
{"peers": ["TGliTmFDTFBLOpyBsled71NjFOZfF3L+w0sdAvcM3xI1nM/
↪ik6NbRzxmwgFBJRZdQ+h2CURQlwxtFxe33U7oldJtK+E1fTk2rOo="]}
```

Then we confirm that Peer 1 has the existing attribute (`my_attribute` from the last step).

```
$ curl http://localhost:14411/identity/pseudonym1/credentials
{"names": [{"name": "my_attribute", "hash": "mtMiZioWORNgV+GeGACsY+rD+lI=", "metadata": {
↪"name": "my_attribute", "schema": "id_metadata", "date": 1593171171.876003}, "attesters
↪": ["TGliTmFDTFBLOg/
↪rrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv+hSjbssLYmps+jlh9rb250LYD7gEH20=
↪"]}]}
$ curl http://localhost:14412/identity/pseudonym2/credentials
{"names": []}
```

1. VERIFICATION REQUEST - Peer 2 will now ask Peer 1 to verify an attribute.

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"hash":
↪"mtMiZioWORNgV+GeGACsY+rD+lI=","value":"dmFsdWU=","schema":"id_metadata"}' "http://
↪localhost:14412/identity/pseudonym2/verify/TGliTmFDTFBLOpyBsled71NjFOZfF3L
↪%2Bw0sdAvcM3xI1nM%2Fik6NbRzxmwgFBJRZdQ%2Bh2CURQlwxtFxe33U7oldJtK%2BE1fTk2rOo%3D"
{"success": true}
```

2. VERIFICATION - Peer 1 finds an outstanding request for verification.

```
$ curl http://localhost:14411/identity/pseudonym1/outstanding/verifications
{"requests": [{"peer": "TGliTmFDTFBLOg/
↪rrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv+hSjbssLYmps+jlh9rb250LYD7gEH20=",
↪ "attribute_name": "my_attribute"}
$ curl -X PUT -H "Content-Type: application/json" -d '{"name":"my_attribute"}' "http://
↪localhost:14411/identity/pseudonym1/allow/TGliTmFDTFBLOg
```

(continues on next page)

(continued from previous page)

```
↪%2Frrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv%2BhSjbssLYmps
↪%2Bjlh9rb25OLYD7gEH20%3D"
{"success": true}
```

3. CHECK - Peer 2 checks the output of the verification process.

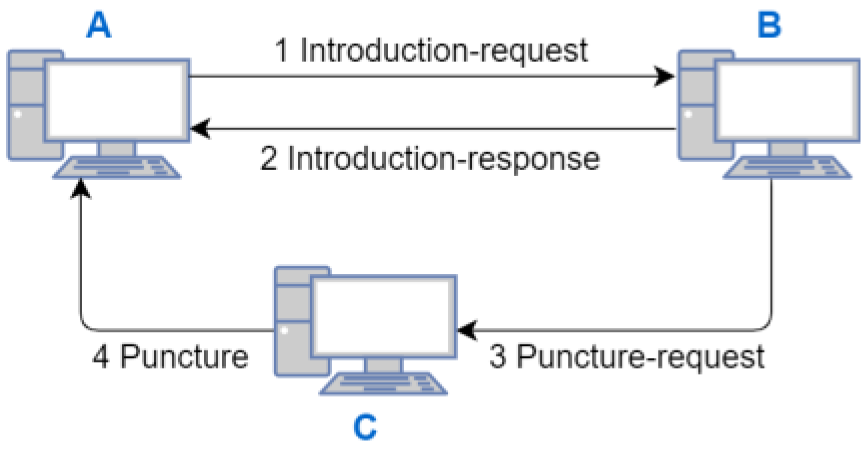
```
$ curl http://localhost:14412/identity/pseudonym2/verifications
{"outputs": [{"hash": "mtMiZioWORNgV+GeGACsY+rD+lI=", "reference": "dmFsdWU=", "match":␣
↪0.9999847412109375}]}
```

6.8 Peer discovery basics

All IPv8 overlays have 4 messages in common: introduction-request, introduction-response, puncture-request, and puncture. These 4 messages are used for peer discovery and NAT puncturing.

The peer discovery protocol runs the following steps in a loop until enough peers have been found:

1. Peer A sends an introduction-request to peer B. Peer B is chosen from an existing pool of neighboring peers.
2. Peer B sends an introduction-response to peer A containing the address of peer C.
3. Peer B sends a puncture-request to peer C containing the address of peer A.
4. Peer C sends a puncture to peer A, puncturing its NAT.



When a peer doesn't yet have a list of neighboring peers, it will select a bootstrap server for peer B. IPv8 bootstrap servers implement the same peer discovery protocol as ordinary peers, except that they respond to introduction-requests for *any* overlay. Once a peer sends an introduction-request to a bootstrap server, the bootstrap server will keep track of the sender and the overlay within which the introduction-request was sent. When sending introduction-responses, the bootstrap server will pick a peer from this list as an introduction candidate (peer C in the image above). Periodically, the bootstrap server will send an introduction-request for a random peer in the list. If the peer doesn't respond with an introduction-response, the bootstrap server will assume that the unresponsive peer is no longer interested in new peers and update its list accordingly.

6.9 Community Best Practices

When working with a small `Community` class you can get away with putting all of your code into the same file. However, as your codebase grows this simple approach becomes unmaintainable (most importantly it becomes hard to test). More formally, whereas you should always start with the KISS principle (**keep it simple, stupid**) we'll show how you can organize your `Community` as it grows larger (using the SOLID design principles).

Pledge: *One of the founding principles of IPv8 is to allow you to program in any style you like. IPv8 should never force your code to fit any particular architecture. This is what separates IPv8 from its predecessor Dispersy.*

6.9.1 SOLID by example

The first letter of the SOLID principles stands for the single-responsibility principle. We'll now discuss what this means for your `Community`.

Any `Community` is in charge of sending and receiving messages between `Peer` instances (identities managed by the `Network` class). A direct implication of this is that any code in your `Community` which is not concerned with handling or sending messages should be extracted. This may be hard to spot, so let's discuss a practical example:

```
class MyCommunity(Community):

    def __init__(self, *args, **kwargs):
        super().__init(*args, **kwargs)
        # ... details omitted ...
        self.last_value = 0
        self.total_value = 0

    @lazy_wrapper(MyPayload)
    def on_my_payload(self, peer, payload):
        self.last_value = payload.value
        self.total_value += payload.value
        self.ez_send(peer, MyResponsePayload(self.total_value))
```

Is there anything wrong with this code? No, and you should always strive to keep your code as simple as possible. However, this style may become unmanageable if your `Community` becomes too big. In this particular example, we see that the `MyCommunity` is storing a state of incoming `payload.value`, which is not its responsibility. This example doesn't follow the SOLID principles and next we'll apply other principles of SOLID to fix it.

Our previous example completely captures and manages the state of `payload.value`. This makes `MyCommunity` a god-class, arguably the worst software engineering anti-pattern. Let's incrementally improve our example. First we'll delegate the incoming information to a specific interface (the I of *interface segregation* in SOLID). The following turns `MyCommunity` into a mediator:

```
class MyCommunity(Community):

    def __init__(self, *args, **kwargs):
        super().__init(*args, **kwargs)
        # ... details omitted ...
        self.value_manager = ValueManager()

    @lazy_wrapper(MyPayload)
    def on_my_payload(self, peer, payload):
        self.value_manager.set_last_value(payload.value)
        self.value_manager.add_to_total(payload.value)
```

(continues on next page)

(continued from previous page)

```
        return_value = self.value_manager.total_value
        self.ez_send(peer, MyResponsePayload(return_value))
```

Has this improved our code? Yes. We can now test all of the methods in `ValueManager` without having to send messages through the `MyCommunity`. Especially if your message handlers are very complex, this can save you a lot of time. This also improves the readability of your code: the `ValueManager` clearly takes care of all value-related state updates. As the responsibility of value-related updates now lies with the `ValueManager`, our `MyCommunity` now again has a single responsibility.

Is our previous improvement perfect? No. We have upgraded our `MyCommunity` from a god-class pattern to a mediator pattern. Our class is still performing low-level operations on the `ValueManager`, violating the dependency inversion principle (the D in SOLID). Dependency inversion consists of both keeping low-level details of dependencies out of a higher-level class and making generic interfaces. You can see that the `MyCommunity` has to call `set_last_value` and `add_to_total`, which are low-level operations. Let's fix that:

```
class MyCommunity(Community):

    def __init__(self, *args, **kwargs):
        super().__init(*args, **kwargs)
        # ... details omitted ...
        self.value_manager = ValueManager()

    @lazy_wrapper(MyPayload)
    def on_my_payload(self, peer, payload):
        return_value = self.value_manager.process(payload.value)
        self.ez_send(peer, MyResponsePayload(return_value))
```

Finally perfection. Our `MyCommunity` no longer has any knowledge of how a `payload.value` is processed. Our `ValueManager` can internally process a value, without knowing about the `payload`. The return value of `ValueManager` is then given back to the `MyCommunity` to send a new message, which is its responsibility. We can still test our `ValueManager` independently, but now also provide our `MyCommunity` with a mocked `ValueManager` to more easily test it.

Some final notes:

- Don't forget that you have `asyncio` at your disposal! You can, for example, give your managers an `asyncio.Future` for you to await.
- You should be wary when applying the Inversion of Control principle to allow your managers to directly send messages from your `Community`. This may violate the dependency inversion principle through your inverted control.

6.9.2 `Community` initialization

To run IPv8 as a service (using `ipv8_service.py`), you need to be able to launch your overlay from user settings (i.e., a configuration dictionary of strings and ints). This conflicts with a dependency injection pattern. A compromise, which is a recurring successful pattern in IPv8, is "create from configuration if not explicitly supplied". In other words, check if a dependency is given to our constructor and create it from the supplied settings if it is not. This is an example:

```
class MyCommunity(Community):

    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        # Create-if-Missing Pattern
        settings = kwargs.pop('settings', MyCommunitySettings())
        if isinstance(settings, dict):
            # Convert user dict to settings object
            settings = MyCommunitySettings.from_dict(settings)

        # Create-if-Missing Pattern
        self.value_manager = kwargs.pop('value_manager',
                                        ValueManager(settings.value_manager))

        super().__init(*args, **kwargs)
```

Note that to pass settings to your overlay it is often better to supply a settings object instead of passing every configuration parameter separately (the latter is known as a *Data Clump* code smell). Passing your settings as an object avoids passing too many arguments to your `Community` (Pylint R0913).

6.10 IPv8 configuration options

The `ipv8/configuration.py` contains the main IPv8 configuration options. IPv8 will read a dictionary that conforms to the configuration format to determine what services to start and which keys to use. By invoking `get_default_configuration()`, you can get a dictionary copy of the default settings for your custom IPv8 configuration.

Table 2: Configuration keys

key	default	description
interfaces		The interfaces to bind to (`UDPIPv4` or `UDPIPv6`) using an IP address and port. If the specified port is blocked, IPv8 will attempt the next free port (up to 10,000 ports over the specified port).
keys		Specify a list of keys, by alias, for IPv8 to use. The curve should be picked from those available in the ECCrypto class. IPv8 will generate a new key if the key file does not exist.
logger		The logger intialization arguments, also see the default Python logger facilities.
walker_interval	0.5	The time interval between IPv8 updates. Each update will trigger all registered strategies to update, mostly this concerns peer discovery.
overlays	[...]	The list of overlay definitions and their respective walking strategies. See the overlay definition section for further details.

6.10.1 Overlay Specifications

Each of the overlay specifications is a dictionary following the following standard:

Table 3: Network overlay definitions

key	description
class	The overlay class to load. Do note that any external overlay definitions will have to be registered in IPv8, see also the overlay creation tutorial.
key	The alias of the key to use for the particular overlay.
walkers	The walker to employ.
bootstrappers	The bootstrappers to use.
initialize	The additional arguments to pass to the constructor of the overlay.
on_start	A list of tuples containing method names and their arguments. These methods are invoked when IPv8 has started.

By default, the `RandomWalk` and `EdgeWalk` strategies are known to IPv8. Respectively these will take care of performing random walks and random walks with reset probability for peer discovery. Each overlay may also specify further custom strategies. Check out the the bootstrapping documentation for more information on configuring bootstrappers per overlay.

By default, IPv8 loads the following overlays:

- DiscoveryCommunity
- HiddenTunnelCommunity
- DHTDiscoveryCommunity

6.10.2 Key Specifications

Each of the key specifications is a dictionary following the following standard:

Table 4: Key definitions

key	description
alias	The name by which this key can be referenced in overlay configuration.
file	The optional file to store the key in.
generation	The curve to use if this key needs to be generated.
bin	The b64 encoded raw key material to use.

It is always required to specify a key `alias`. If you specify a `file` IPv8 will attempt to load your key from this file. Only if the file does not exist, will the `generation` or `bin` be referenced. If a `file` has been specified, once a key has been loaded it will be written to the specified `file`. If you specify a `bin`, IPv8 will prefer to use this raw key material over generating a new key from the key curve specified by `generation`. You must provide IPv8 with at least one of the key source material options (a `file`, a `bin` or a `generation`) to have a valid key configuration.

6.11 IPv8 bootstrapping

Peers discover each other through other Peers, as specified in the the Peer discovery basics. We call this type of Peer discovery *introduction*. However, you cannot be introduced to a new Peer if you don't know anyone to introduce you in the first place. This document discusses how IPv8 provides you your first contact.

6.11.1 Bootstrap servers (rendezvous nodes)

To provide you an initial point of introduction, IPv8 mainly uses bootstrap servers (also known as rendezvous nodes). You can connect to these servers to attempt to find other Peers for your overlay. A bootstrap server will then respond to your request for Peers with Peers that have contacted it for the same overlay. However, these bootstrap servers only introduce you to other Peers, they do not (and can not) actually join any overlay. A bootstrap server also has no way of knowing what your overlay does, it only knows its 20-byte identifier.

By default, IPv8 comes supplied with a handful of bootstrap IP addresses and bootstrap DNS addresses. You can freely extend or replace the default servers with your own. To tell IPv8 what bootstrap IP addresses and bootstrap DNS addresses it should connect to, you can use the `DispersyBootstrapper` class.

Using bootstrap servers

To have IPv8 load a bootstrapper for your overlay, you can simply add it to your `ConfigBuilder.add_overlay()` step. This is the easiest way to load a bootstrapper. For most intents and purposes you can simply use `default_bootstrap_defs` provided by `ipv8.configuration` (see the overlay tutorial). However, you can also completely change the bootstrap servers you use. For example, this code sets two bootstrap addresses (the IP address 1.2.3.4 with port 5 and the DNS address tribler.org with port 5):

```
ConfigBuilder().add_overlay("MyCommunity",
                            "my id",
                            [WalkerDefinition(Strategy.RandomWalk, 42, {
↪'timeout': 3.0})],
                            [BootstrapperDefinition(Bootstrapper.
↪DispersyBootstrapper,
                                                   {'ip_addresses': [('1.2.3.4
↪', 5)],
                                                    'dns_addresses': [(
↪'tribler.org', 5)]})],
                            {},
                            [])
```

If you are using the `loader` instead of `configuration` you can load a bootstrapper into your launcher as follows:

```
@overlay('my_module.some_submodule', 'MyCommunity')
@walk_strategy(RandomWalk)
@bootstrapper(DispersyBootstrapper, kw_args={'ip_addresses': [('1.2.3.4', 5)],
↪'dns_addresses': [('tribler.org', 5)]})
class MyLauncher(CommunityLauncher):
    pass
```

If you are using neither the `loader` nor the `configuration`, you can manually add a `DispersyBootstrapper` instance to your overlay's `bootstrappers` field. However, you should do so before the overlay starts discovering Peers (i.e., before *IPv8.start()* is invoked).

Running a bootstrap server

To run your own bootstrap server simply call `scripts/tracker_plugin.py`. For example, this code will attempt to bind to port 12345:

```
python tracker_plugin.py --listen_port=12345
```

You are responsible to configure your port forwarding over your hardware in such a way that the specified listen port is connectable.

6.11.2 UDP broadcasts

Next to bootstrap servers, IPv8 also allows serverless Peer discovery for LANs. This is done using UDP broadcast sockets. These broadcasts consist of sending an IPv8 probe UDP packet to all possible ports of the IPv4 broadcast address. You should consider this option if you are having trouble connecting to other Peers in your home network. This method is usually very effective for simple home networks. However, complex home (or work) network setups may still fail to discover these local Peers.

Using UDP broadcasts

Loading UDP broadcast bootstrapping for your overlay functions much the same as using bootstrap servers. Again, you can simply add it to your `ConfigBuilder.add_overlay()` step:

```
ConfigBuilder().add_overlay("MyCommunity",
                            "my id",
                            [WalkerDefinition(Strategy.RandomWalk, 42, {
↪'timeout': 3.0})],
                            [BootstrapperDefinition(Bootstrapper.
↪UDPBroadcastBootstrapper, {})],
                            {},
                            [])
```

If you are using the `loader` instead of `configuration` you can load a bootstrapper into your launcher as follows:

```
@overlay('my_module.some_submodule', 'MyCommunity')
@walk_strategy(RandomWalk)
@bootstrapper(UDPBroadcastBootstrapper)
class MyLauncher(CommunityLauncher):
    pass
```

If you are using neither the `loader` nor the `configuration`, you can manually add a `UDPBroadcastBootstrapper` instance to your overlay’s `bootstrappers` field.

6.11.3 Making your own Bootstrapper

As you may have noticed, loading a `DispersyBootstrapper` and a `UDPBroadcastBootstrapper` is highly similar. This is because they both inherit from the `Bootstrapper` interface. You can fulfill the same interface to provide your own bootstrapping mechanism. To use custom bootstrappers you will either have to use the `loader` or manual loading methods.

6.12 Key generation options

The `ipv8/keyvault/crypto.py` file contains the main public key cryptography class for IPv8: `ECCrypto`. It allows you to generate the following keys:

Table 5: Available curves for key generation

name	curve	backend
very-low	SECT163K1	M2Crypto
low	SECT233K1	M2Crypto
medium	SECT409K1	M2Crypto
high	SECT571R1	M2Crypto
curve25519	EC25519	Libsodium

The `M2Crypto` backend keys do not actually use the `M2Crypto` backend, but use a `python-cryptography` backend. These `M2Crypto` curves are supported for backwards compatibility with the Dispersy project. For new applications, only the `curve25519` should be used.

Generally you will create either a new `ECCrypto` instance (if you wish to modify or extend the base cryptography) or use the default `default_eccrypto` instance. The following methods are most commonly used:

- `generate_key()`: generate a new key from a given curve name.
- `key_to_bin()`: serialize a given key into a string.
- `key_from_private_bin()`: load a private key from a string.
- `key_from_public_bin()`: load a public key from a string.

The following methods will usually be handled by IPv8 internally:

- `key_to_hash()`: convert a key into a `sha1` string (usually accessed through `Peer.mid`).
- `create_signature()`: create a signature for some data (usually handled by the `Community` class).
- `is_valid_signature()`: checks a signature for validity (usually handled by the `Community` class).

6.13 Message serialization

IPv8 gives you as much control as possible over the messages you send over the Internet. The `Overlay` (or `Community`) class lets you send arbitrary strings over the (UDP) `endpoint`. However efficient this may be, having non-standardized string contruction for each message of your overlay can distract from the overal overlay design. This is the age-old dichotomy of maintainable versus performant code.

The basic class for serializing objects from and to strings/network packets is the `Serializer` (`ipv8/messaging/serialization.py`). Though the `Serializer` is extensible, you will mostly only need the default serializer `default_serializer`. You can use the `Serializer` with classes of the following types:

Table 6: Serializable classes

class	path	description
Serializable	ipv8/messaging/serializatioı	Base class for all things serializable. Should support the instance method to_pack_list() and the class method from_unpack_list().
Payload	ipv8/messaging/payload.py	Extension of the Serializable class with logic for pretty printing.
VariablePayload	ipv8/messaging/lazy_payloa	Less verbose way to specify Payloads, at the cost of performance.
dataclass	ipv8/messaging/payload_da	Use dataclasses to send messages, at the cost of control and performance.

Other than the `dataclass`, each of these serializable classes specifies a list of primitive data types it will serialize to and from. The primitive data types are specified in the *data types* Section. Each serializable class has to specify the following class members (`dataclass` does this automatically):

Table 7: Serializable class members

member	description
format_list	A list containing valid data type primitive names.
names	Only for VariablePayload classes, the instance fields to bind the data types to.

As an example, we will now define four completely wire-format compatible messages using the four classes. Each of the messages will serialize to a (four byte) unsigned integer followed by an (two byte) unsigned short. If the `dataclass` had used normal `int` types, these would have been two signed 8-byte integers instead. Each instance will have two fields: `field1` and `field2` corresponding to the integer and short.

```python
class MySerializable(Serializable):
    format_list = ['I', 'H']

    def __init__(self, field1: int, field2: int) -> None:
        self.field1 = field1
        self.field2 = field2

    def to_pack_list(self) -> list[tuple]:
        return [('I', self.field1),
                ('H', self.field2)]

    @classmethod
    def from_unpack_list(cls: type[MySerializable],
                         field1: int, field2: int) -> MySerializable:
        return cls(field1, field2)


class MyPayload(Payload):
    format_list = ['I', 'H']

    def __init__(self, field1: int, field2: int) -> None:
        self.field1 = field1
        self.field2 = field2
```

(continues on next page)

(continued from previous page)

```
    def to_pack_list(self) -> list[tuple]:
        return [('I', self.field1),
                ('H', self.field2)]

    @classmethod
    def from_unpack_list(cls: type[MyPayload],
                         field1: int, field2: int) -> MyPayload:
        return cls(field1, field2)


class MyVariablePayload(VariablePayload):
    format_list = ['I', 'H']
    names = ['field1', 'field2']


@vp_compile
class MyCVariablePayload(VariablePayload):
    format_list = ['I', 'H']
    names = ['field1', 'field2']


I = type_from_format('I')
H = type_from_format('H')


@dataclass
class MyDataclassPayload:
    field1: I
    field2: H
```

To show some of the differences, let's check out the output of the following script using these definitions:

```
serializable1 = MySerializable(1, 2)
serializable2 = MyPayload(1, 2)
serializable3 = MyVariablePayload(1, 2)
serializable4 = MyCVariablePayload(1, 2)
serializable5 = MyDataclassPayload(1, 2)

print("As string:")
print(serializable1)
print(serializable2)
print(serializable3)
print(serializable4)
print(serializable5)
```

```
As string:
<__main__.MySerializable object at 0x7f732a23c1f0>
MyPayload
| field1: 1
| field2: 2
MyVariablePayload
```

(continues on next page)

(continued from previous page)

```
| field1: 1
| field2: 2
MyCVariablePayload
| field1: 1
| field2: 2
MyDataclassPayload
| field1: 1
| field2: 2
```

6.13.1 Datatypes

Next to the unsigned integer and unsigned short data types, the default Serializer has many more data types to offer. The following table lists all data types available by default, all values are big-endian and most follow the default Python `struct` format. A `Serializer` can be extended with additional data types by calling `serializer.add_packer(name, packer)`, where `packer` represent the object responsible for (un)packing the data type. The most commonly used packer is `DefaultStruct`, which can be used with arbitrary `struct` formats (for example `serializer.add_packer("I", DefaultStruct(">I"))`).

Table 8: Available data types

member	bytes	unserialized type
?	1	boolean
B	1	unsigned byte
BBH	4	[unsigned byte, unsigned byte, unsigned short]
BH	3	[unsigned byte, unsigned short]
c	1	signed byte
f	4	signed float
d	8	signed double
H	2	unsigned short
HH	4	[unsigned short, unsigned short]
I	4	unsigned integer
l	4	signed long
LL	8	[unsigned long, unsigned long]
q	8	signed long long
Q	8	unsigned long long
QH	10	[unsigned long long, unsigned short]
QL	12	[unsigned long long, unsigned long]
QQHHBH	23	[unsigned long long, unsigned long long, unsigned short, unsigned short, unsigned byte, unsigned long]
ccB	3	[signed byte, signed byte, unsigned byte]
4SH	6	[str (length 4), unsigned short]
20s	20	str (length 20)
32s	20	str (length 32)
64s	20	str (length 64)
74s	20	str (length 74)
c20s	21	[unsigned byte, str (length 20)]
bits	1	[bit 0, bit 1, bit 2, bit 3, bit 4, bit 5, bit 6, bit 7]
ipv4	6	[str (length 7-15), unsigned short]
raw	?	str (length ?)
varlenBx2	1 + ? * 2	[str (length = 2), ...] (length < 256)

continues on next page

Table 8 – continued from previous page

member	bytes	unserialized type
varlenH	2 + ?	str (length ? < 65356)
varlenHutf8	2 + ?	str (encoded length ? < 65356)
varlenHx20	2 + ? * 20	[str (length = 20), ...] (length < 65356)
varlenH-list	1 + ? * (2 + ??)	[str (length < 65356)] (length < 256)
varlenI	4 + ?	str (length < 4294967295)
doublevarlenH	2 + ?	str (length ? < 65356)
payload	2 + ?	Serializable

Some of these data types represent common usage of serializable classes:

Table 9: Common data types

member	description
4SH	(IP, port) tuples
20s	SHA-1 hashes
32s	libnacl signatures
64s	libnacl public keys
74s	libnacl public keys with prefix

Special instances are the `raw` and `payload` data types.

- `raw`: can only be used as the last element in a format list as it will consume the remainder of the input string (avoid if possible).
- `payload`: will nest another `Serializable` instance into this instance. When used, the `format_list` should specify the class of the nested `Serializable` and the `to_pack_list()` output should give a tuple of `("payload", the_nested_instance)`. The `VariablePayload` automatically infers the `to_pack_list()` for you. See the `NestedPayload` class definition for more info.

6.13.2 The ez_pack family for Community classes

All subclasses of the `EZPackOverlay` class (most commonly subclasses of the `Community` class) have a short-cut for serializing messages belonging to the particular overlay. This standardizes the prefix and message ids of overlays. Concretely, it uses the first 23 bytes of each packet to handle versioning and routing (demultiplexing) packets to the correct overlay.

The `ezr_pack` method of `EZPackOverlay` subclasses takes an (integer) message number and a variable amount of `Serializable` instances. Optionally you can choose to not have the message signed (supply the `sig=True` or `sig=False` keyword argument for respectively a signature or no signature over the packet).

The `lazy_wrapper` and `lazy_wrapper_unsigned` decorators can then respectively be used for unserializing payloads which are signed or not signed. Simply supply the payload classes you wish to unserialize to, to the decorator.

As some internal messages and deprecated messages use some of the message range, you have the messages identifiers from 0 through 234 available for your custom message definitions. Once you register the message handler and have the appropriate decorator on the specified handler method your overlay can communicate with the Internet. In practice, given a `COMMUNITY_ID` and the payload definitions `MyMessagePayload1` and `MyMessagePayload2`, this will look something like this example (see the overlay tutorial for a complete runnable example):

```python
class MyCommunity(Community):
    community_id = COMMUNITY_ID
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, settings: CommunitySettings) -> None:
        super().__init__(settings)

        self.add_message_handler(1, self.on_message)

    @lazy_wrapper(MyMessagePayload1, MyMessagePayload2)
    def on_message(self, peer: Peer, payload1: MyMessagePayload1,
                   payload2: MyMessagePayload2) -> None:
        print("Got a message from:", peer)
        print("The message includes the first payload:\n", payload1)
        print("The message includes the second payload:\n", payload2)

    def send_message(self, peer: Peer) -> None:
        packet = self.ezr_pack(1, MyMessagePayload1(), MyMessagePayload2())
```

It is recommended (but not obligatory) to have single payload messages store the message identifier inside the `Payload.msg_id` field, as this improves readability:

```
    self.add_message_handler(MyMessage1, self.on_message1)
    self.add_message_handler(MyMessage2, self.on_message2)
    self.ez_send(peer, MyMessage1(42))
    self.ez_send(peer, MyMessage2(7))
```

If you are using the `@dataclass` wrapper you can specify the message identifier through an argument instead. For example, `@dataclass(msg_id=42)` would set the message identifier to `42`.

Of course, IPv8 also ships with various `Community` subclasses of its own, if you need inspiration.

6.13.3 Using external serialization options

IPv8 is compatible with pretty much all third-party message serialization packages. However, before hooking one of these packages into IPv8 you may want to ask yourself whether you have fallen victim to marketing hype. After all, `XML` is the one unifying standard we will never switch away from, right? Oh wait, no, it's `JSON`. My bad, it's `Protobuf`. Or was it `ASN.1`? You get the point. In this world, only the core `IPv8` serialization format remains constant.

There are three main ways to hook in external serialization: *per message*, *per Serializer* and *per Community*. The three methods can be freely mixed.

Custom serialization per message

If you only want to use custom seralization for (part of) a single overlay message, you can use `VariablePayload` field modification (this also works for dataclass payloads). This method involves implementing the methods `fix_pack_<your field name>` and `fix_unpack_<your field name>` for the fields of your message that use custom serialization. Check out the following example:

```
@vp_compile
class VPMessageKeepDict(VariablePayload):
    msg_id = 1
    format_list = ['varlenH']
    names = ["dictionary"]
```

(continues on next page)

(continued from previous page)

```
    def fix_pack_dictionary(self, the_dictionary: dict) -> bytes:
        return json.dumps(the_dictionary).encode()

    @classmethod
    def fix_unpack_dictionary(cls: type[VPMessageKeepDict],
                              serialized_dictionary: bytes) -> dict:
        return json.loads(serialized_dictionary.decode())


@dataclass(msg_id=2)
class DCMessageKeepDict:
    dictionary: str

    def fix_pack_dictionary(self, the_dictionary: dict) -> str:
        return json.dumps(the_dictionary)

    @classmethod
    def fix_unpack_dictionary(cls: type[DCMessageKeepDict],
                              serialized_dictionary: str) -> dict:
        return json.loads(serialized_dictionary)
```

In both classes we create a message with a single field `dictionary`. To pack this field, we use `json.dumps()` to create a string representation of the dictionary. When loading a message, `json.loads()` is used to create a dictionary from the serialized data. Instead of `json` you could also use any serialization of your liking.

Using the same transformations for all fields makes your payloads very lengthy. In this case, you may want to look into specifying a custom serialization format.

Custom serialization formats

It is possible to specify new formats by adding packing formats to a `Serializer` instance. You can easily do so by overwriting your `Community.get_serializer()` method. This `Serializer` is sandboxed per `Community` instance, so you don't have to worry about breaking other instances. Check out the following example and note that the message is now much smaller at the expense of having to define a custom (complicated) packing format.

```
@vp_compile
class Message(VariablePayload):
    msg_id = 1
    format_list = ['json', 'json', 'json', 'json']
    names = ["d1", "d2", "d3", "d4"]


class PackerJSON(Packer):

    def pack(self, data: Any) -> bytes:
        packed = json.dumps(data).encode()
        size = struct.pack(">H", len(packed))
        return size + packed

    def unpack(self, data: bytes, offset: int,
               unpack_list: list, *args: Any) -> int:
        size, = struct.unpack_from(">H", data, offset)
```

(continues on next page)

(continued from previous page)

```
        json_data_start = offset + 2
        json_data_end = json_data_start + size

        serialized = data[json_data_start:json_data_end]
        unpack_list.append(json.loads(serialized))

        return json_data_end


class MyCommunity(Community):

    def get_serializer(self) -> Serializer:
        serializer = super().get_serializer()
        serializer.add_packer('json', PackerJSON())
        return serializer
```

The line `serializer.add_packer('json', PackerJSON())` adds the new format `json` that is used in `Message`. In fact, any further message added to this `Community` can now use the `json` format. However, you may also note some additional complexity in the `PackerJSON` class.

Our custom packer `PackerJSON` implements two required methods: `pack()` and `unpack()`. The former serializes data using custom serialization (`json.dumps()` in this case). We use a big-endian unsigned short (`">H"`) to determine the length of the serialized JSON data. The `unpack()` method creates JSON objects from the serialized data, returning the new offset in the `data` stream and adding the object ot the `unpack_list` list.

Custom Community data handling

It is possible to circumvent IPv8 message formats altogether. In its most extreme form, you can overwrite `Community.on_packet(packet)` to inspect all raw data sent to your `Community` instance. The `packet` is a tuple of `(source_address, data)`. You can write raw data back to an address using `self.endpoint.send(address, data)`.

If you want to mix with other messages, you should use the message byte. The following example shows how to use JSON serialization without any IPv8 serialization. Note that we need to do our own signature checks now.

```
class MyCommunity(Community):
    community_id = os.urandom(20)

    def __init__(self, settings: CommunitySettings) -> None:
        super().__init__(settings)
        self.event = None
        self.add_message_handler(1, self.on_message)

    def send_message(self, peer: Peer) -> None:
        message = json.dumps({"key": "value", "key2": "value2"})
        public_key = to_hex(self.my_peer.public_key.key_to_bin())
        signature = to_hex(self.my_peer.key.signature(message.encode()))

        signed_message = json.dumps({"message": message,
                                     "public_key": public_key,
                                     "signature": signature}).encode()
```

(continues on next page)

(continued from previous page)

```
        self.endpoint.send(peer.address,
                           self.get_prefix() + b'\x01' + signed_message)

    def on_message(self, source_address: Address, data: bytes) -> None:
        # Account for 1 byte message id
        header_length = len(self.get_prefix()) + 1
        # Strip the IPv8 multiplexing data
        received = json.loads(data[header_length:])

        public_key = self.crypto.key_from_public_bin(unhexlify(received["public_key"]))
        valid = self.crypto.is_valid_signature(public_key,
                                               received["message"].encode(),
                                               unhexlify(received["signature"]))
        self.logger.info("Received message %s from %s, the signature is %s!",
                         received['message'], source_address, valid)
```

6.13.4 Nested Payloads

It is possible to put a `Payload` inside another `Payload`. We call these nested payloads. You can specify them by using the `"payload"` datatype and setting the `Payload` class in the format list. For a `VariablePayload` this looks like the following example.

```
class A(VariablePayload):
    format_list = ['I', 'H']
    names = ["foo", "bar"]


class B(VariablePayload):
    format_list = [A, 'H']  # Note that we pass the class A
    names = ["a", "baz"]
```

For dataclass payloads this nesting is supported by simply specifying nested classes as follows.

```
@dataclass(msg_id=1)
class Message:
    @dataclass
    class Item:
        foo: int
        bar: int

    item: Item
    items: [Item]  # Yes, you can even make this a list!
    baz: int
```

6.14 Advanced attestation service usage

This document assumes you have a basic understanding of identification flows in IPv8, as documented in the overlay tutorial. This document addresses the following three topics:

- Enabling anonymization.
- Setting a REST API access token.
- Setting a rendezvous token.

Each of these features can be enabled independently, but should all be enabled when dealing with actual identity data.

6.14.1 Enabling anonymization

Purpose: *disallow device fingerprinting.*

In the basic identity tutorial we created the following configuration:

```
for peer_id in [1, 2]:
    configuration = get_default_configuration()
    configuration['keys'] = [
        {'alias': "anonymous id", 'generation': "curve25519", 'file': f"keyfile_{peer_id}
↪.pem"}]
    configuration['working_directory'] = f"state_{peer_id}"
    configuration['overlays'] = []
```

To enable anonymization of all traffic through the identity layer we need to load the anonymization overlay. This is done by editing the loaded overlays through `configuration['overlays']`, as follows:

```
for peer_id in [1, 2]:
    configuration = get_default_configuration()
    configuration['keys'] = [
        {'alias': "anonymous id", 'generation': "curve25519", 'file': f"keyfile_{peer_id}
↪.pem"}]
    configuration['working_directory'] = f"state_{peer_id}"
    configuration['overlays'] = [overlay for overlay in configuration['overlays']
                                 if overlay['class'] == 'HiddenTunnelCommunity']
```

Inclusion of the `'HiddenTunnelCommunity'` overlay automatically enables anonymization of identity traffic. Note that this anonymization:

1. Requires other peers to be online.
2. Requires additional startup time. It can take several seconds for an anonymized circuit to be established.

6.14.2 Setting a REST API key

Purpose: *disallow other local services from hijacking IPv8.*

In the basic identity tutorial we started the REST API as follows:

```
# Start the IPv8 service
ipv8 = IPv8(configuration)
await ipv8.start()
rest_manager = RESTManager(ipv8)
await rest_manager.start(14410 + peer_id)
```

To set a REST API key, we will have to pass it to the `RESTManager` constructor, as follows (replacing `"my secret key"` with your key):

```
# Start the IPv8 service
ipv8 = IPv8(configuration)
await ipv8.start()
rest_manager = RESTManager(ipv8)
await rest_manager.start(14410 + peer_id, api_key="my secret key")
```

All requests to the core will then have to use either:

1. The `X-Api-Key` HTTP header set to the key value (this is the preferred option).
2. A URL parameter `apikey` set to the key value.

Any HTTP request without either of these entries or using the wrong key will be dropped.

6.14.3 Using a REST API X509 certificate

Purpose: *provide transport layer security (TLS) for the REST API.*

In the basic identity tutorial we started the REST API as follows:

```
# Start the IPv8 service
ipv8 = IPv8(configuration)
await ipv8.start()
rest_manager = RESTManager(ipv8)
await rest_manager.start(14410 + peer_id)
```

To use a certificate file, we will have to pass it to the `RESTManager` constructor, as follows (replacing `cert_file` with the file path of your certificate file for the particular IPv8 instance):

```
ipv8 = IPv8(configuration)
await ipv8.start()
ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
ssl_context.load_cert_chain(cert_file)
rest_manager = RESTManager(ipv8)
await rest_manager.start(14410 + peer_id, ssl_context=ssl_context)
```

This can (and should) be combined with an API key. Also note that if you start two IPv8 instances, you would normally want them to have different certificates.

If you don't have a certificate file, you can generate one with `openssl` as follows:

```
openssl req -newkey rsa:2048 -nodes -keyout private.key -x509 -days 365 -out certfile.pem
cat private.key >> certfile.pem
rm private.key
```

6.14.4 Setting a rendezvous token

Purpose: *disallow exposure of pseudonym public keys.*

In the basic identity tutorial we used HTTP requests without a rendezvous token as follows:

```
$ curl http://localhost:14411/identity/pseudonym1/peers
{"peers": ["TGliTmFDTFBLOg/
↪rrouc7qXT1ZKxHFvzxb4IVRYDPdbN4n7eFFuaT385YNW4aoh3Mruv+hSjbssLYmps+jlh9rb250LYD7gEH20=
↪"]}
```

By including the X-Rendezvous header entry and setting it to a shared secret in base64 encoding, we can guide a rendezvous between peers. The following is an example of a rendezvous using the shared identifier string abc.

```
$ curl --header "X-Rendezvous: YWJj" http://localhost:14411/identity/pseudonym1/peers
```

Notes:

- Include this header in all of your requests.
- If you want to switch rendezvous tokens, first call identity/{pseudonym_name}/unload.
- Any identifier over 20 bytes is truncated.
- You may still find peers other than those you are interested in if you happen to share the same rendezvous identifier. Always communicate and verify the public key of your counterparty beforehand (use the identity/{pseudonym_name}/public_key REST endpoint for this).

6.15 Advanced peer discovery configuration

This document assumes you have a basic understanding of peer discovery in IPv8, as documented in the peer discovery basics. This document covers the configuration of built-in DiscoveryStrategy classes, explained in the network IO and the DiscoveryStrategy tutorial. In particular, we address the following three topics:

- How the RandomWalk works and when to use it.
- How the EdgeWalk works and when to use it.
- How to parameterize the walk strategies in relation to the overlays they operate on.
- How the RandomChurn works and when to use it.

6.15.1 The Random walk strategy

The "Random" walk strategy finds new peers by randomly querying existing peers. Every IPv8 tick a random peer is chosen from the overlay that a `RandomWalk` class is attached to. The chosen peer is then asked for new peers (using the introduction-request mechanism), to which it responds with a random peer that it knows. In case a random peer knows no other peers, it responds with no peer. In the case that no peers are available to ask for introductions, the bootstrap mechanism is invoked.

The most important aspect of the "Random" walk strategy is that **it is not a search strategy**. Because there is no attempt to avoid previously queried peers, the mechanism is very simple. However, a previously queried peer that has no new available connections is not avoided, making the "Random" walk waste bandwidth if there are no further peers to find. Secondly, if a strong clique forms in the connection graph it can take very long to find new peers if networks get large. Nevertheless, the strategy **is robust against networks with high node churn**. For the purposes of IPv8 connections the "Random" strategy is usually the preferred strategy as the number of connected peers remains relatively low.

6.15.2 The Edge walk strategy

The "Edge" walk strategy is another strategy to find new peers based on existing peers. However, unlike the "Random" strategy, the "Edge" strategy is a search strategy. The "Edge" strategy uses a depth-first search through known peers. Each edge of a search starts at a peer found by the bootstrap process. Peers are then queried in order of introduction for a random peer they are connected to. After a configured search depth has been reached, a new edge is constructed.

The "Edge" walk is typically used to find as many peers as possible at the cost of additional overhead. Peers are not often revisited and therefore this strategy is bandwidth efficient. Of course, managing the edges means that there is more computational overhead in managing peers. Finding many peers is rarely preferential for "normal" IPv8 use, but it can be useful when creating network crawlers.

6.15.3 Parameterizing walk strategies

For every walk strategy IPv8 maintains a number of target peers and for every overlay IPv8 a number of maximum peers. The former tells IPv8 to keep calling the walker until a certain number of peers has been accepted and the latter tells IPv8 when to stop accepting new connections. **You should never ever configure the walk strategies to fill out an overlay's maximum peers.** By doing so, you can inadvertently create network partitions.

The default settings, of `20` target peers for the walk and `30` maximum connections for the overlay, are tried and tested. More connections do not imply a better overlay and may, in fact, prove detrimental. For example, connecting to `1000` peers may completely overload a machine's buffers and drop most of the incoming UDP packets. There are three main points to remember. Firstly, it is generally better to depend on a good information dissemination strategy than on a large number of connections. Secondly, the number of connections you open has diminishing returns for the speed at which information makes its way through your network. Finally, the more connections you open, the higher the chance that attackers come into connection with a peer.

It is a good idea to first run extensive experiments before changing walk parameterization. Finding peers is complex because it is (a) a process that uses asynchronous messaging, (b) the connection graph between peers is highly dynamic, and (c) peers may not have the same parameterization.

If you do decide to change the parameterization, do not mix overlays with different parameterizations. Sampling from a set with replacement is a statistical process. IPv8 does not come with a built-in mechanism to balance out the sampling rate according to differences in set sizes (i.e., the number of peers in each overlay). If you do have different walk parameterizations, `max_peers`, or `target_peers`, make sure to give each overlay a separate `Network` instance.

6.15.4 The RandomChurn churn strategy

The “Random” churn strategy is a strategy to remove unresponsive peers from overlays. Peers are randomly sampled and if they have not been communicated with in the last thirty seconds, they will be pinged for a response. If a peer does not respond to a ping request, it will be removed from the `Network`.

If you have a shared `Network` instance, you only need to apply the `RandomChurn` to a single overlay. However, be cautioned that you need a `RandomChurn` for each `Network` instance.

By default the `RandomChurn` strategy checks eight peers simultaneously. A total three pings are sent, one every ten seconds, after 30 seconds of inactivity. These settings should be adjusted if a large number of peers are being connected to.

6.16 TunnelCommunity

This document contains a description of how to use the `TunnelCommunity` class for anonymous communication and the `HiddenTunnelCommunity` extension of this functionality, which provides service-based end-to-end anonymous communication. The `TunnelCommunity` class should be used when passively contributing to the network health and the `HiddenTunnelCommunity` should be used when you want to actively send over the network.

In particular this document will **not** discuss how creating and managing circuits and setting up hidden services works, for this we refer the reader to the Tor research.

6.16.1 Interface

The `HiddenTunnelCommunity` class provides an interface for anonymous messaging. The most important methods of this class are:

- `build_tunnels()`: which will start making the required amount of circuits for the given hop count
- `active_data_circuits()`: will return the currently available circuits to send data over
- `tunnels_ready()`: the quotient of available circuits to send over, compared to `settings.min_circuits`
- `tunnel_data()`: which will send data to some destination through an exit node
- `register_service()`: which provide end-to-end connections for a particular service

Note that you will have to call `build_tunnels()` before sending any data. For example `build_tunnels(1)` will construct 1 hop circuits to maintain a circuit pool of a size between `settings.min_circuits` and `settings.max_circuits`. Initializing these settings will be covered in a later section.

You can then call `tunnel_data()` to send data through a circuit (and eventually an exit node) to some destination. The circuit to send over should be one of the circuits returned by active_data_circuits(). The destination can be any ip + port tuple. The message type should be `u"data"` and finally the payload can be an arbitrary message. For example one could send the packet 1 to `("1.2.3.4", 5)`:

```
community.tunnel_data(community.active_data_circuits()[0], ("1.2.3.4", 5), u"data", "1")
```

6.16.2 Hidden services

The HiddenTunnelCommunity class can also find other circuit exit points for a particular service identifier. In other words, it is possible to anonymously find other peers communicating anonymously. To do this, some sort of DHT mechanism needs to be registered (currently Tribler uses mainline DHT). Whatever DHT mechanism is chosen should be placed in the dht_provider attribute of the HiddenTunnelCommunity.

Whenever a peer is found for a service, or rather the anonymizing relay for a peer, the callback defined through register_service() is called with the available peers. You can then use this peer to tunnel_data() to. The complete example is given below:

```
def on_peer(peer):
    community.tunnel_data(community.active_data_circuits()[0], peer, u"data", "my␣
↪message to this peer")
community.register_service("my service identifier", hops=1, on_peer)
```

6.16.3 TunnelSettings

The community can be initialized using the TunnelSettings class. This class contains the following fields:

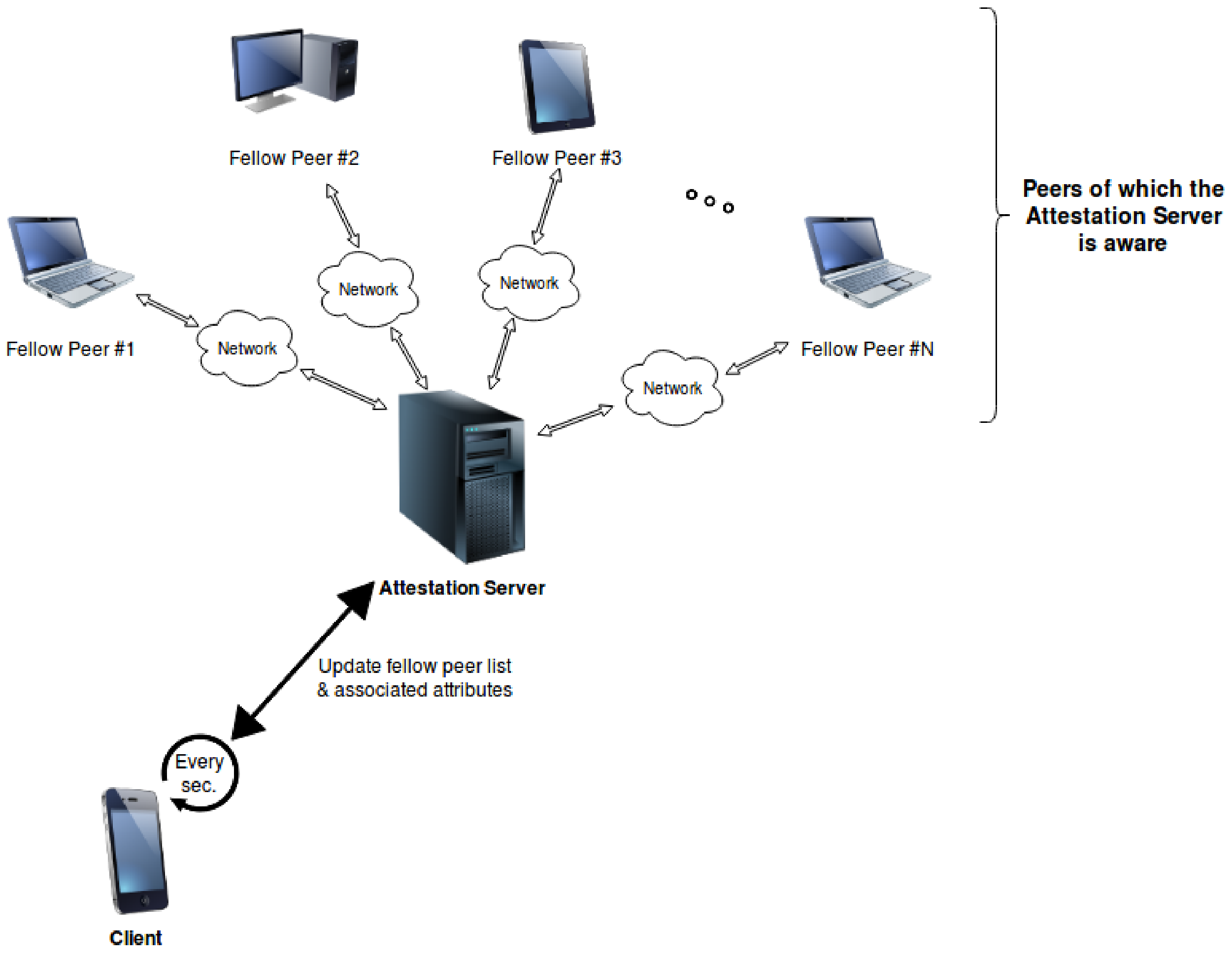
- *min_circuits*: the minimum amount of circuits to create before tunnels_ready() gives a value larger than 1.0
- *max_circuits*: the maximum amount of circuits to create
- *max_joined_circuits*: the maximum amount of circuits, which are not our own, we will partake in
- *max_time*: the time after which a circuit will be removed (for security reasons)
- *max_time_inactive*: the time after which an idle circuit is removed
- *max_traffic*: the amount of traffic after which a circuit will be removed (for security reasons)
- *become_exitnode*: whether or not we will exit data for others (may have legal ramifications)

6.17 Attestation Prototype Documentation

This document describes the original design of the IPv8-based Android application and credential authority.

6.17.1 Passive Updates in the IPv8 Android Application:

Example behaviour of *passive* and *regular* updates in an **IPv8 Android Application** instance, regarding fellow peer lists and their associated attributes. It should be noted that the **Attestation Server** is nothing more than a *well-known peer* itself, accessed through an HTTP URL (127.0.0.1:8086/attestation):



Every second, the application will do the following:

1. Forward a (**GET**: *peers*) request to a certain peer, with the aim of updating the local list of known peers.
2. When the response for the previous request arrives, containing unique identifiers (IDs) for the fellow peers, update the local list of known peers. Simultaneously, forward a (**GET**: *attributes*) for each peer ID, in order to obtain a list of the transaction names and hashes as found in the latest 200 blocks associated to the peer. The information in these responses is stored locally as well.
3. Forward a (**GET**: *attributes*) request to and for the peer contacted in **Step 1**. Similarly to **Step 2**, the response should contain the transaction names and hashes as found in the latest 200 blocks associated to the peer. This information is stored locally.

6.17.2 Flow of Communication in Active Attestation Requests:

Example RESTful based communication flow for serving **Attestation Requests**, as currently implemented in the demo application:

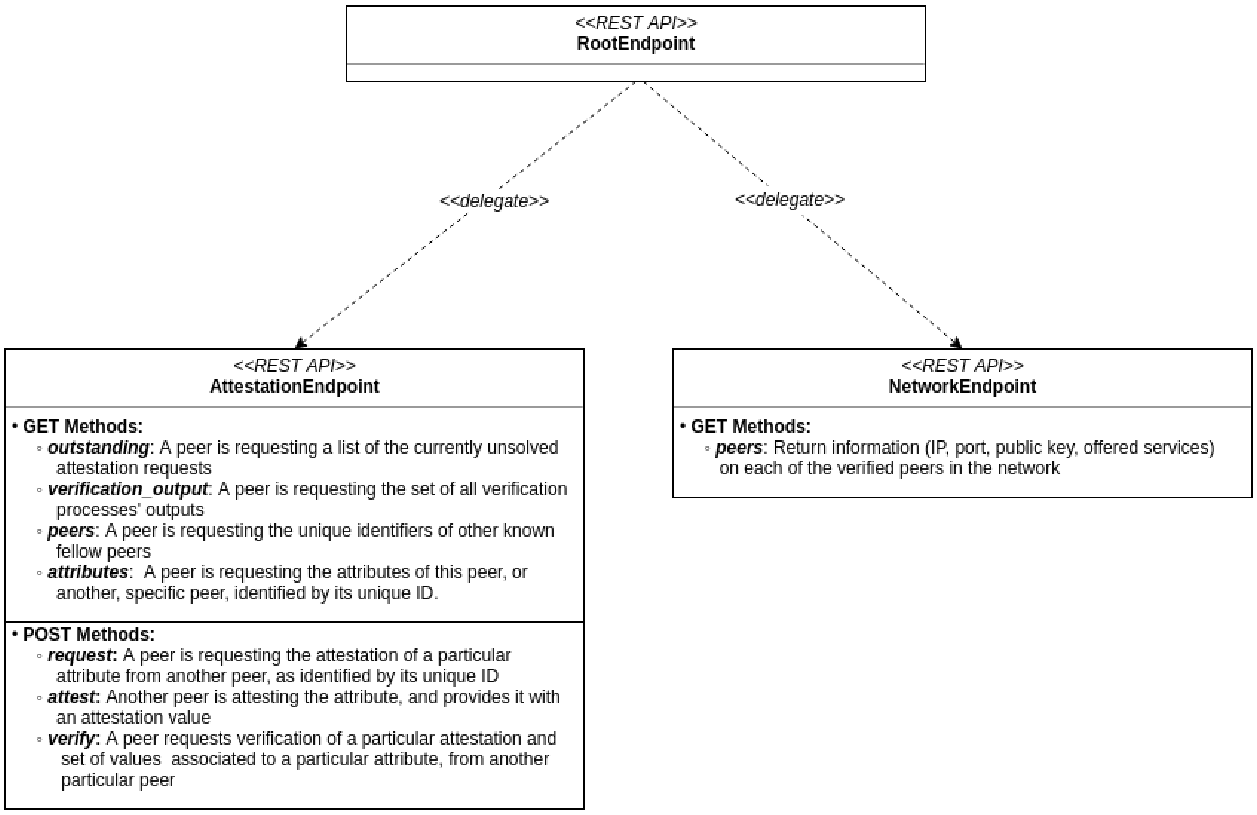
1. The **Client**'s user interacts with their device in order to **obtain attestation** for a particular **attribute**.
2. The **Client** attempts to update the list of known peers by forwarding a (**GET**: *peers*) request to a well-known fellow peer (the **Attestation Server**).

3. The **Client** demands **attestation** for the **attribute**, by forwarding a (**POST**: *request*) request for the particular **attribute** to every known peer, as identified by their unique ID. In this particular demo implementation, only the well-known peer (the **Attestation Server**) is forwarded the request.
4. A **Fellow Peer** will await for at least one other fellow peer (the **Client**) to be present in the network, before accepting and solving attestation requests. It will do so by repeatedly forwarding (**GET**: *peers*) requests to the well-known peer, until this returns a non-empty list of peers.
5. The **Fellow Peer** will probe for **outstanding** (i.e. as of yet unresolved) **attestation requests**. To obtain such a list, it forwards a (**GET**: *outstanding*) request to the well-known peer. The request will return a list of tuples of the form `(<b64_encoded_requesting_peer_ID>, <attribute_name>)`, each representing an **outstanding request**.
6. The **Fellow Peer** chooses to solve the request, and **attest** the requested **attribute**. This process is done locally, in the peer itself. The result of the attestation is encoded in *Base64* format, and packed in a (**POST**: *attest*) request, which is forwarded to the well-known peer (the **Attestation Server**).
7. The well-known peer (the **Attestation Server**), returns the **attestation** to the **Client**, as a response to their original (**POST**: *request*) **attestation request**.
8. The **Client** receives the **attestation**, and displays the **successful attestation** status to the device user.

6.17.3 Peers and their REST APIs:

It should be mentioned that the protocol, which abides to the REST paradigms, used towards facilitating communication between peers, in this particular implementation, is the **Hypertext Transfer Protocol (HTTP)**.

The diagram below describes the **REST API** of an **IPv8** object (in this current implementation version). These are requests which can be handled by the **IPv8** objects:


<<REST API>>
RootEndpoint
<<delegate>>
<<delegate>>
<<REST API>>
AttestationEndpoint
• GET Methods:
◦ outstanding: A peer is requesting a list of the currently unsolved attestation requests
◦ verification_output: A peer is requesting the set of all verification processes' outputs
◦ peers: A peer is requesting the unique identifiers of other known fellow peers
◦ attributes: A peer is requesting the attributes of this peer, or another, specific peer, identified by its unique ID.
• POST Methods:
◦ request: A peer is requesting the attestation of a particular attribute from another peer, as identified by its unique ID
◦ attest: Another peer is attesting the attribute, and provides it with an attestation value
◦ verify: A peer requests verification of a particular attestation and set of values associated to a particular attribute, from another particular peer
<<REST API>>
NetworkEndpoint
• GET Methods:
◦ peers: Return information (IP, port, public key, offered services) on each of the verified peers in the network

Below, a detailed explanation of the REST Requests is presented:

- **GET Methods** sent to the **AttestationEndpoint**:
  - `type = "outstanding"` : Retrieve a list of tuples of the form `(<b64_encoded_requesting_peer_ID>, <attribute_name>)`, each representing an, as of yet, unresolved attestation request (from a peer identified by `<b64_encoded_requesting_peer_ID>` for the attribute `<attribute_name>`).
  - `type = "verification_output"` : Retrieve a dictionary (equivalently, a map) of the form: `<b64_attribute_name_hash> -> (<b64_attribute_value_hash>, <confidence_value>)`. The dictionary should contain a record of all the previous verification processes' results.
  - `type = "peers"` : Retrieve a list of Base64 encoded peer IDs, which uniquely identify the peers in the **first Identity Overlay**.
  - `type = "attributes"` : Return a list of tuples of the form `(<transaction_name>, <b64_transaction_hash>)`, from the the latest 200 blocks associated with a **particular peer**.

    Optional parameters :

    * `mid = <b64_peer_mid>` : Identifies the **particular peer** (using its unique ID) whose attributes are being requested. If this parameter is missing, the **peer** shall default to the target endpoint's peer.
- **POST Methods** sent to the **AttestationEndpoint**:
  - `type = "request"` : Request the attestation of a particular attribute from a particular peer.

    Required additional parameters :

    * `mid = <b64_peer_mid>`: Identifies the peer (by its unique ID) from which the attestation is requested.
    * `attribute_name = <attribute_name>`: The name of the attribute, for which attestation is requested.
  - `type = "attest"` : Attest a particular attribute, for a particular peer. Additionally, return an attested value for the aforementioned attribute.

    Required additional parameters :

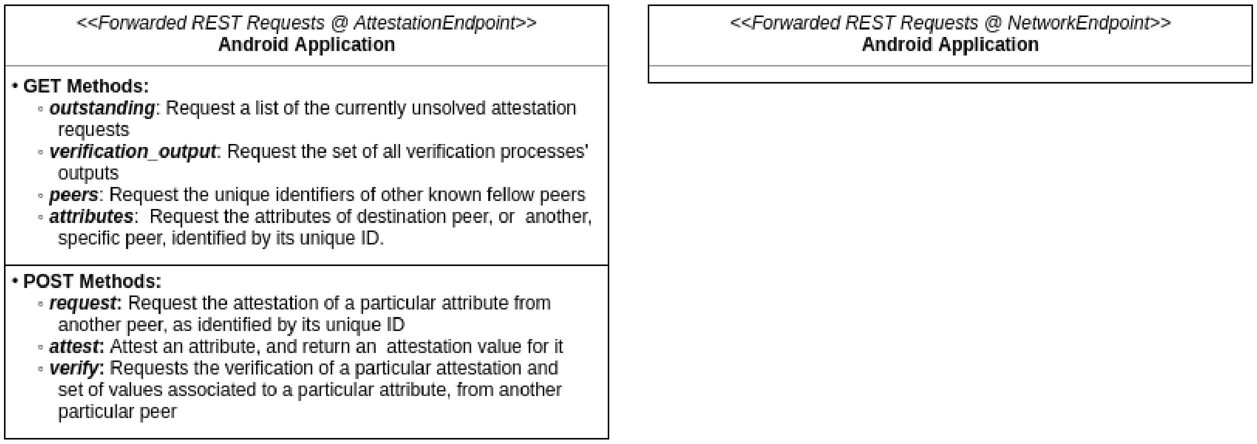
    * `mid = <b64_peer_mid>`: Identifies the peer (by its unique ID) from which the attestation is requested.
    * `attribute_name = <attribute_name>`: The name of the attribute, for which attestation is requested.
    * `attribute_value = <b64_attribute_value>`: Base64 encoded attested value for the attribute.
  - `type = "verify"` : Request the verification of a particular attestation and set of values associated to a particular attribute, from another particular peer.

    Required additional parameters :

    * `mid = <b64_peer_mid>`: Identifies the peer (by its unique ID) from which the attestation is requested.
    * `attribute_hash = <b64_attribute_hash>`: A Base64 encoded hash of the attribute's particular attestation, subject to verification.
    * *attribute_values = <b64_attribute_values_string>*: A variable length list of Base64 encoded attribute values, for which verification is required. *<b64_attribute_values_string>* is a comma separated string of values, that is: `<b64_attribute_values_string> = "<b64_attr_val_1>, <b64_attr_val_2>,...,<b64_attr_val_N>"`, where `N` may be arbitrarily large.

- **GET Methods** sent to the **NetworkEndpoint**:
  - **Any GET request**: Retrieve a dictionary (equivalently, a map) holding information on each verified peer in the network. The dictionary's structure will be the following: `<b64_peer_mid> -> {"id": <peer_IP>, "port": <peer_port>, "public_key": <b64_peer_public_key>, "services": <b64_peer_services_list>}`. `<b64_peer_services_list>` is a list which identifies the known services supported by the peer. The dictionary itself is returned within a dictionary: `{"peers": <peers_info_dictionary>}`.

The diagram below describes the **REST Requests** implemented in the **Android Application** (in this current implementation version). These requests are forwarded to, and handled by the **IPv8** object's **REST APIs**:



<<Forwarded REST Requests @ AttestationEndpoint>>
Android Application

- **GET Methods:**
  - ***outstanding***: Request a list of the currently unsolved attestation requests
  - ***verification_output***: Request the set of all verification processes' outputs
  - ***peers***: Request the unique identifiers of other known fellow peers
  - ***attributes***: Request the attributes of destination peer, or another, specific peer, identified by its unique ID.
- **POST Methods:**
  - ***request***: Request the attestation of a particular attribute from another peer, as identified by its unique ID
  - ***attest***: Attest an attribute, and return an attestation value for it
  - ***verify***: Requests the verification of a particular attestation and set of values associated to a particular attribute, from another particular peer

<<Forwarded REST Requests @ NetworkEndpoint>>
Android Application

All requests must abide to the specifications detailed above (in this section).

6.17.4 Attestation Process: Detailed Explanation

Below is a detailed description of the general flow for handling **incoming attestation requests**. It should be noted that this description is general, and does not necessarily refer to this particular implementation version (i.e. of the demo). Moreover, the reader shoould also note that **attestation** is requested from a particular peer (see documentation above for (**POST**): *attest* @ **AttestationEndpoint**), however, the actual HTTP request is forwarded to another peer, potentially different from the **attester** itself. Hence, there shall be two cases for the **attestation request**:

1. When a peer receives the HTTP request, packs it, and forwards it to the peer from which attestation is requested (i.e. the **attester** peer).
2. When the peer from which attestation is requested (i.e. the **attester** peer) receives the attestation request itself, as obtained from the 1st case.

The two cases shall be further described in the sections to follow. To better understand what each of them do, however, we must introduce two callback methods of the `AttestationCommunity` class, located in the **community.py** module. The two callbacks are stored in the `self.attestation_request_callbacks` field, which is an array that should contain 2 elements, intuitively, each being a callback. The methods which set the callbacks are described in the following (both are located in the `AttestationCommunity` class):

- *set_attestation_request_callback(f)*: this is the method which stores the function *f* in *self.attestation_request_callbacks[0]* field. *f* is the method actually handles the **attestation**. This method is called in *AttestationCommunity*'s *on_request_attestation* method when some peer requests the attestation from us (i.e. we are the **attester**). *f* must return a string if the attestation is made, this represents the attestation value. If *None* is returned, however, the attestation is not made. As input, *f* receives the following:
  - `<sender_peer>: Peer`: A `Peer` object representing the peer which forwarded this request to us.

    - `<attribute_name>`: `string`: A `string` object, representing the name of the attribute requiring attestation.

- *set_attestation_request_complete_callback(f)*: this is the method which stores the function *f* in *self.attestation_request_callbacks[1]* field. *f* is called back when an attestation has been completed.`f` needn't return anything. It should accept the following input parameters:
    - `<sender_peer>`: `Peer`: A `Peer` object representing the peer which forwarded this request to us.
    - `<attribute_name>`: `string`: A `string` object, representing the name of the attribute requiring attestation.
    - `<attestation_hash>`: `string`: A `string` object, representing the hash of the attestation blob.
    - `<signer_peer> = None`: `string`: A `Peer` object representing the signer of the attestation (i.e. usually us).

Below is a description of the two scenarios of the attestation process presented above. **Make sure to also read this section's last subsection. It contains details on how the attestation process works for this demo implementation**.

Receiving the HTTP Attestation Request

In the **community.py** module, and the `AttestationCommunity` class, the method `request_attestation` is called by the `render_POST` method in the `AttestationEndpoint` class, upon receiving a (**POST**: *request*) HTTP request. Attestation might be required from the peer receiving this request, or from another peer. Regardless, the `request_attestation` method will do the following:

1. It will add the request to its own *request cache*.
2. It will create some **metadata** from the request, that is: the *attribute's name* to be attested, and the request's *public key*.
3. It will create 3 payloads: an **authentication payload** (communication security between this peer and the **attester**), the actual **attestation request payload** (created from the metadata), and the **time distribution payload** (created from the global time). The payloads are packed together in a packet which is forwarded to the attester.

It should be noted that this method should normally be called once, when a raw (**POST**: *request*) HTTP request is received. This method will send a packet (with the contents as described above) directly to the **attester** peer, which might, in fact, be this peer, or another different peer.

Receiving the Attestation Request in the Attester Peer

In the **community.py** module and the `AttestationCommunity` class, the method `on_request_attestation` is called when attestation is requested from us. That is, when an attestation packet, as created by the `request_attestation` method in the `AttestationCommunity` is received. The `on_request_attestation` method will do the following:

1. Unpack the packet, and get the 3 payloads: **attestation payload**, **request payload**, and **time distribution payload**. The source of the request is seen as the peer which last forwarded the request.
2. The attestation payload's metadata is retrieved and is used towards the attestation process. Attestation is performed by calling the `self.attestation_request_callbacks[0]` with the *source peer* and *attribute name* as parameters. If a value is returned, attestation has been performed, otherwise attestation has not been performed.
3. A blob is created from the attestation value. The blob is first used as a parameter in the `self.attestation_request_callbacks[1]`, together with the *source peer*, and the *attribute name*. After returning from the aforementioned call, the attestation blob is sent to the *source peer* (i.e. the peer from which we received this request).

Side Note: Detailed Attestation Flow in the Demo

This case is a bit curious, since the peer forwarding the (**POST**: *attest*) HTTP request (i.e. the peer running in the **main.py** script, in the method `make_attribute`), is indeed solving the **attestation request**, however, it is not actually holding it. In the demo, the peer holding the request is the well-known peer behind `localhost:8086/attestation` URL. The agent running in the **main.py** script is made aware of the request by having previously forwarded a (**GET**: *outstanding*) HTTP request to the well-known peer. It will serve the attestation request by forwarding a (**POST**: *attest*) request back to the `localhost:8086/attestation` (well-known) peer.

As previously mentioned, the well-known peer behind `localhost:8086/attestation` is actually holding the **attestation requests**. For each request forwarded to this peer, a `Deferred` object is created, which is attached to a `yield`. This is done in the `on_request_attestation` method (of the `AttestationEndpoint` class), which is called as `self.attestation_request_callbacks[0]` in `on_request_attestation` method (of the `AttestationCommunity` class). The `yield` will suspend the thread, which will now wait for the `Deferred` object to yield something. For this object to yield something, someone has to call the `Deferred.callback(<value>)` method on it. This is done when the agent behind **main.py** forwards a (**POST**: *attest*) HTTP request to the well-known peer. In the `render_POST` method of `AttestationEndpoint` class, for this `type` of request, the deferred object is retrieved, and called back with an attestation value (obtained from the (**POST**: *attest*) request) as the <value> parameter. This allows the `Deferred` object to yield this value, and, in turn, allows the `on_request_attestation` (of `AttestationCommunity`) method to continue executing, which was halted by its call to `self.attestation_request_callbacks[0]`, i.e. `on_request_attestation` (of `AttestationEndpoint`). The `on_request_attestation` (of `AttestationCommunity`) eventually calls `self.attestation_request_callbacks[1]`, i.e. `on_attestation_complete`, and finally sends back the attestation to the source peer.

6.17.5 Annex: General Notes on Demo Classes

Notes on the IPv8 application

Below is a brief description of some of the important classes in the IPv8 application (which are employed in the demo):

- The `RootEndpoint` class will be the one which receives the requests from the peers. This class will delegate work to its children. Currently, the children are: `AttestationEndpoint` and `NetworkEndpoint`. All these classes must subclass the BaseEndpoint class. The request chain is modeled like a tree, where the `RootEndpoint` is the root of the tree and the `AttestationEndpoint` and `NetworkEndpoint` classes are its children (and implicitly leaves) . When they are added, they are associated a `path`, which is the `endpoint` field of the request (e.g. *attestation* or *network*). This is how the root knows to delegate requests to the right children (e.g. `<node_ip>:<port>/<path>`).
- The `NetworkEndpoint` handles the *peer discovery* requests. It is (currently) a simple class which has the simple task of returning a list of verified nodes in the system and some relevant information on them, such as: *IP*, *port*, *public key*, and *offered services*. However, the Android Phone Application does not forward requests (GET or POST) to this endpoint.
- The `AttestationEndpoint` class handles a set of both **POST** and **GET** requests. It features a more complex set of methods than `NetworkEndpoint`.
- The `Response` class is used to model HTTP responses. It will essentially just define a response (status_code, content) which will then be sent back as a response.
- The `RESTManager` class handles the start and stop of the server's HTTP API service. On startup, the class will create a `RootEndpoint` object, which will implicitly create the two objects: a `NetworkEndpoint` object and an `AttestationEndpoint` object. It will then attach itself to incoming request from `localhost` (`127.0.0.1`) at port `8085`. To create a `RestManager` object, one must submit a *session* object, which is in fact of the `IPv8` class type (this latter instantiation requires a *configuration* object).

Notes on the Python Scripts

The roles.py script

- Holds the class `TestRESTAPI` which is a subclass of `RESTManager`. It does the exact same thing as the `RESTManager` class, with the singular difference that it listens for incoming request on the `8086` port.
- The function `sleep(time)` is a simple NOOP function, which essentially forces the thread to sleep for a user specified amount of time (set by the parameter `time`), since it forces it to wait for a deferred to yield.
- The `create_working_dir(path)` function creates a working directory at the specified path, which will hold a number of **.pem** files, and an **sqlite** database (currently, for attestation and identity matters).
- The `initialize_peer(path)` function is used to create a peer (of type `TestRESTAPI`). It will generate a default peer configuration, which it will then modify to set the network overlays, which in this case is a *AttestationCommunity* and an *IdentityCommunity*. It will also set a peer discovery strategy within each of those overlays, which in this case is a *RandomWalk* with a limit of 20 peers and timeout of 60 seconds. A working directory is constructed, as described above (`create_working_dir(*)`). From this *configuration* an `IPv8` object is created, which is in turn used to create the peer, i.e. the `TestRESTAPI`. This will also be assigned a (blank) HTML interface at `URL = 127.0.0.1:8086/attestation`, that will define an access point to its REST API. The URL will be returned as the second object of the return statement.
- The `stop_peers(*peers)` function will simply do its best to stop the list of peers passed as parameter to it.
- The `make_request(url, type, arguments={})` function forwards a request of the specified `type` to the specified `url`, with the specified set of `arguments`. In this particular example, we usually have`url = '127.0.0.1:8086/attestation'`, (**GET**) (for **peers** and **outstanding** attestation requests) and (**POST**) (for attesting outstanding requests). This function uses an `Agent` object, which models a simple HTTP client, that is able to generate HTTP requests given a set of parameters. Requests forwarded by this method are attached a callback, which is able to read the HTTP response body.

The main.py script

- This script does not define any classes. It only defines a set of functions.
- This script creates a peer object by calling **roles.py**'s `initialize_peer(path)`, with the `path = "attester"`, and retrieves its URL towards accessing its REST API (`idowner_restapi = 127.0.0.1:8086/attestation`).
- The `wait_for_peers(idowner_restapi)` function will forward a (**GET**: *peers*) request towards the `idowner_restapi` URL, i.e. the peer located behind it. The methods seeks to discover other peers in the network. It recursively calls itself every 4 seconds. Upon identification of at least one peer, it will stop calling itself, and return the list of discovered peers.
- The `wait_for_attestation_request(attester_restapi)` function will forward a (**GET**: *outstanding*) towards the `idowner_restapi` URL, i.e. the peer located behind it. This methods seeks to find any, as of yet, unresolved *attestation* requests. It will recursively call itself every 4 seconds. Upon identification of at least one outstanding request, it will stop calling itself, and return the list of discovered requests.
- The `make_attribute()` method implements the main logic of the **main.py** script. It is triggered by a 500 millisecond deferred call . The method's flow is the following: It fist awaits for at least one fellow peer to show up by calling `wait_for_peers(idowner_restapi)`, where the `idowner_restapi` is the URL of the `TestRESTAPI` object created as a global variable within the script. After this, it will await for at least one request to appear, by calling `wait_for_attestation_request(attester_restapi)`, where the `idowner_restapi` is the URL of a `TestRESTAPI` object created as a global variable within the script. It will then solve each request by forwarding a (**POST**: *attest*) with the requested attribute ("**QR**") and a random encoded value associated to this

attribute (since it is a demo) back to the `TestRESTAPI` object via its public URL. Once it finishes solving the requests it had initially found, it will end its loop and terminate its activity.

Notes on the Android Application

The MainActivity class

The following outlines the main behaviour of this class

- Every second, the following happens:
  - The application will probe for peers. It does so by forwarding a (**GET**: *peers*) to a well-known peer. If successful, the request will yield a response with an updated list of peers (represented by their unique IDs). Response timing is arbitrary, and cannot be anticipated clearly since it depends on network and other factors. Regardless, the peer will attempt to employ a local data structure, which is used for storing unique peer IDs (this might have been updated in the meantime, but it is not certain). If not possible to access the aforementioned list, it will return an empty list. It should be noted that when peer IDs are retrieved, the application will try to retrieve and store their attributes as well (by forwarding (**GET**: *attributes*), with an additional request parameter `mid = <peer_mid>`).
  - Next, it will attempt to request (and if the response arrives on time, update) the well-known peer's attributes (by forwarding a (**GET**: *attributes*), **WITHOUT** the additional request parameter `mid = <peer_mid>`). Generally, however, this step refers to retrieving the local attributes list of the known peers (which should have been obtained as a collateral of the peer discovery process, as described above). If the list is not in use, it will return it, otherwise, an empty list is returned. If the list is empty, then the button, which demands attestation, is turned red, otherwise it is turned green.
- Asynchronously, the user may press the button on the interface, which will require that the 'QR' attribute be attested. If the application has discovered peers (i.e. at least one), then it will forward an attestation request (for this attribute) at every known peer in the network (by forwarding (**POST**: *request*) HTTP requests, with additional request parameters `mid = <peer_mid>`, and `attribute_name = "QR"`). Although attestation is requested from specific peers, in this demo implementation, any peer can solve them (see section **Side Note: Detailed Attestation Flow in the Demo**). All requests are forwarded to the well-known peer.

The SingleShotRequest class

This class models a simple HTTP request, whose destination is hard-coded to be `localhost:8085`. The constructor allows one to specify the endpoint (currently, we can have *attestation* or *network*, the latter not being used in the current Application implementation). It is also possible to specify the method (e.g. *POST*, *GET*, *PUT*, etc.), and a collection of (key, value) tuples, which act as the request's parameters. It should be noted that one specifies the nature of the request as part of the latter parameter, using the `('type', <req_type>)` tuple (`<req_type>` can be *outstanding*, *verification_output*, *peers*, *attributes*, etc.).

The AttestationRESTInterface class

This class defines a set of public methods, which handle the creation and transmission of the different types of currently available HTTP requests. The following request types are defined (all of which are forwarded to the **attestation** endpoint):

- **GET Method:**
  - `retrieve_peers`: Forward a request for the peers in the system. Upon receiving a response, a callback is invoked to store the mid's of the discovered peers. This method also forwards requests for retrieving the peers' attributes, and relies of the callback to handle their storing.

  - retrieve_outstanding: Forward a request for the outstanding attestation requests in the system. Upon receiving a response, a callback is invoked to locally store the collection of discovered pending attestation requests.
  - retrieve_verification_output: Forward a request to get (all) the results of the verification processes. Upon receiving a response to the request, a callback is invoked which stores the verification results locally.
  - retrieve_attributes(String mid): Forward a request for the attributes of a particular peer in the system, as identified by their mid. On response, invoke a callback, and store the attributes of the peer locally.
  - retrieve_attributes(void): Forward a request for the attributes of the peer this request is sent to. On response, invoke a callback, which stores the attributes of the peer locally.
- **POST Method:**
  - put_request: Forward a request which asks for attestation of a particular attribute from a particular peer.
  - put_attest: Attests a particular value of a particular attribute for a particular peer.
  - put_verify: Forward a request which asks for verification from a particular peer of a particular set of values associated to a particular attribute.

6.18 Using the IPv8 attestation service

This document assumes you have a basic understanding of network overlays in IPv8, as documented in the overlay tutorial. You will learn how to use the IPv8 attestation *HTTP REST API*. This tutorial will use curl to perform HTTP GET and POST requests.

Note that this tutorial will make use of the Python IPv8 service. An Android binding is also available (including demo app).

6.18.1 Running the IPv8 service

Fill your main.py file with the following code (runnable with python3 main.py):

```python
from asyncio import run
from base64 import b64encode

from ipv8.configuration import get_default_configuration
from ipv8.REST.rest_manager import RESTManager
from ipv8.util import run_forever
from ipv8_service import IPv8


async def start_communities() -> None:
    # Launch two IPv8 services.
    # We run REST endpoints for these services on:
    #  - http://localhost:14411/
    #  - http://localhost:14412/
    # This script also prints the peer ids for reference with:
    #  - http://localhost:1441*/attestation?type=peers
    for i in [1, 2]:
        configuration = get_default_configuration()
        configuration['logger']['level'] = "ERROR"
```

(continues on next page)

(continued from previous page)

```
        configuration['keys'] = [
            {'alias': "anonymous id", 'generation': "curve25519", 'file': "ec%d_
↪multichain.pem" % i},
        ]

        # Only load the basic communities
        requested_overlays = ['DiscoveryCommunity', 'AttestationCommunity',
↪'IdentityCommunity']
        configuration['overlays'] = [o for o in configuration['overlays'] if o['class']␣
↪in requested_overlays]

        # Give each peer a separate working directory
        working_directory_overlays = ['AttestationCommunity', 'IdentityCommunity']
        for overlay in configuration['overlays']:
            if overlay['class'] in working_directory_overlays:
                overlay['initialize'] = {'working_directory': 'state_%d' % i}

        # Start the IPv8 service
        ipv8 = IPv8(configuration)
        await ipv8.start()
        rest_manager = RESTManager(ipv8)
        await rest_manager.start(14410 + i)

        # Print the peer for reference
        print("Starting peer", b64encode(ipv8.keys["anonymous id"].mid))

    await run_forever()


run(start_communities())
```

Running the service should yield something like the following output in your terminal:

```
$ python3 main.py
Starting peer aQVwz9aRMRypGwBkaxGRSdQs80c=
Starting peer bPyWPyswqXMhbW8+0RS6xUtNJrs=
```

You should see two messages with 28 character base64 encoded strings. These are the identifiers of the two peers we launched using the service. You can use these identifiers for your reference when playing around with sending attestation requests. In your experiment you will see other identifiers than the `aQVwz9aRMRypGwBkaxGRSdQs80c=` and `bPyWPyswqXMhbW8+0RS6xUtNJrs=` shown above.

As a sanity check you can send your first HTTP `GET` requests and you should see that each peer can at least see the other peer. Note that you might find more peers in the network.

```
$ curl http://localhost:14411/attestation?type=peers
["bPyWPyswqXMhbW8+0RS6xUtNJrs="]
$ curl http://localhost:14412/attestation?type=peers
["aQVwz9aRMRypGwBkaxGRSdQs80c="]
```

6.18.2 Functionality flows

Generally speaking there are two (happy) flows when using the IPv8 attestation framework. The first flow is the enrollment of an attribute and the second flow is the verification of an existing/enrolled attribute. Both flows consist of a distinct set of requests (and responses) which we will explain in detail in the remainder of this document.

To test a flow, we start the two peers we created previously. If you did not remove the key files (`*.pem`) after the first run, you will start the same two peers as in the last run. In our case the output of starting the service is as follows:

```
$ python main.py
Starting peer aQVwz9aRMRypGwBkaxGRSdQs80c=
Starting peer bPyWPyswqXMhbW8+0RS6xUtNJrs=
```

In our case this means that peer `aQVwz9aRMRypGwBkaxGRSdQs80c=` exposes its REST API at `http://localhost:14411/` and peer `bPyWPyswqXMhbW8+0RS6xUtNJrs=` exposes its REST API at `http://localhost:14412/`. If you did not modify the ports in the initial scripts, you will have two different peer identifiers listening at the same ports. For convenience we will refer to our first peer as *Peer 1* and our second peer as *Peer 2*.

As a last note, beware of URL encoding: when passing these identifiers they need to be properly formatted (+ and = are illegal characters). In our case we need to use the following formatting of the peer identifiers in URLs (for Peer 1 and Peer 2 respectively):

```
aQVwz9aRMRypGwBkaxGRSdQs80c%3D
bPyWPyswqXMhbW8%2B0RS6xUtNJrs%3D
```

Enrollment/Attestation flow

To enroll, or attest, an attribute we will go through the following steps:

1. Sanity checks: Peer 1 and Peer 2 can see each other and have no existing attributes.
2. Peer 1 requests attestation of an attribute by Peer 2.
3. Peer 2 attests to the requested attribute.
4. Peer 1 checks its attributes to confirm successful attestation.

0. SANITY CHECK - First we check if both peers can see each other using their respective interfaces.

```
$ curl http://localhost:14411/attestation?type=peers
["bPyWPyswqXMhbW8+0RS6xUtNJrs="]
$ curl http://localhost:14412/attestation?type=peers
["aQVwz9aRMRypGwBkaxGRSdQs80c="]
```

Then we confirm that neither peer has existing attributes. Note that `http://*:*/attestation?type=attributes` is shorthand for `http://*:*/attestation?type=attributes&mid=mid_b64` where the identifier is equal to that of the calling peer.

```
$ curl http://localhost:14411/attestation?type=attributes
[]
$ curl http://localhost:14412/attestation?type=attributes
[]
```

1. ATTESTATION REQUEST - Peer 1 will now ask Peer 2 to attest to an attribute.

```
$ curl -X POST "http://localhost:14411/attestation?type=request&mid=bPyWPyswqXMhbW8
↪%2B0RS6xUtNJrs%3D&attribute_name=my_attribute"
```

2. ATTESTATION - Peer 2 finds an outstanding request for attestation. Peer 2 will now attest to some attribute value of Peer 1 (`dmFsdWU%3D` is the string `value` in base64 encoding).

```
$ curl http://localhost:14412/attestation?type=outstanding
[["aQVwz9aRMRypGwBkaxGRSdQs80c=", "my_attribute", "e30="]]
$ curl -X POST "http://localhost:14412/attestation?type=attest&
↪mid=aQVwz9aRMRypGwBkaxGRSdQs80c%3D&attribute_name=my_attribute&attribute_value=dmFsdWU
↪%3D"
```

3. CHECK - Peer 1 confirms that he now has an attested attribute.

```
$ curl http://localhost:14411/attestation?type=attributes
[["my_attribute", "oEkkmxqu0Hd/aMVpSOdyP0SIlUM=", {"name": "my_attribute", "schema": "id_
↪metadata", "date": 1592227939.021873}, "bPyWPyswqXMhbW8+0RS6xUtNJrs="]]
$ curl http://localhost:14412/attestation?type=attributes
[]
```

Attribute verification flow

To verify an attribute we will go through the following steps:

1. Sanity checks: Peer 1 and Peer 2 can see each other and Peer 1 has an existing attribute.
2. Peer 2 requests verification of an attribute of Peer 1.
3. Peer 1 allows verification of its attribute.
4. Peer 2 checks the verification output for the requested verification.

0. SANITY CHECK - First we check if both peers can see each other using their respective interfaces.

```
$ curl http://localhost:14411/attestation?type=peers
["bPyWPyswqXMhbW8+0RS6xUtNJrs="]
$ curl http://localhost:14412/attestation?type=peers
["aQVwz9aRMRypGwBkaxGRSdQs80c="]
```

Then we confirm that Peer 1 has the existing attribute (`my_attribute` from the last step).

```
$ curl http://localhost:14411/attestation?type=attributes
[["my_attribute", "oEkkmxqu0Hd/aMVpSOdyP0SIlUM=", {}, "bPyWPyswqXMhbW8+0RS6xUtNJrs="]]
$ curl http://localhost:14412/attestation?type=attributes
[]
```

1. VERIFICATION REQUEST - Peer 2 will now ask Peer 1 to verify an attribute.

```
$ curl -X POST "http://localhost:14412/attestation?type=verify&
↪mid=aQVwz9aRMRypGwBkaxGRSdQs80c%3D&attribute_hash=oEkkmxqu0Hd%2FaMVpSOdyP0SIlUM%3D&
↪attribute_values=dmFsdWU%3D"
```

2. VERIFICATION - Peer 1 finds an outstanding request for verification.

```
$ curl http://localhost:14411/attestation?type=outstanding_verify
[["bPyWPyswqXMhbW8+0RS6xUtNJrs=", "my_attribute"]]
$ curl -X POST "http://localhost:14411/attestation?type=allow_verify&mid=bPyWPyswqXMhbW8
↪%2B0RS6xUtNJrs%3D&attribute_name=my_attribute"
```

3. CHECK - Peer 2 checks the output of the verification process.

```
$ curl http://localhost:14412/attestation?type=verification_output
{"oEkkmxqu0Hd/aMVpSOdyP0SIlUM=": [["dmFsdWU=", 0.9999847412109375]]}
```

CHAPTER SEVEN

SEARCH

- genindex
- modindex
- search