Cicada Documentation

Release 0.2.1-alpha

George

Oct 23, 2019

Contents

Cicada is a resilient communication framework with peer-to-peer routing.

Features:

- Lower bandwidth requirements for service providers
- Highly-efficient and resilient routing between users
- Safe & secure encryption among trusted peers
- Improved user performance

CHAPTER 1

Installation

There are a few minor dependencies that are easily `pip`able; the biggest requirement is that `pygame` is used in the visualization tools:

```
$ pip install -r requirements.txt
```

If you want to build the documentation as well, install the `full_requirements.txt`, which contains all of the Sphinx dependencies.

There are multiple ways of interacting with the *Cicada* library:

- The `cicada.py` script is a command-line interface for both creating a swarm and joining an existing swarm. You define a runtime configuration that executes commands in sequence. See the *Runtime Interpreter* documentation for details.
- The `visualizer.py` script is a visualizer that lets you arbitrarily connect a swarm of peers, watch them exchange messages, and stabilize. See the *Visualization section* for controls.
- The `samples/` directory holds a handful of applications for the library, one of which is a single-room chatting app.

Unfortunately, the library is currently only available on Linux (and possible OS X) because of the dependencies I use for NAT traversal (specifically, pynetinfo). I'll be looking into a cross-platform solution soon.

1.1 Using Cicada in Your Application

Cicada comes with sample applications, but its up to you to use the library to create a peer-to-peer application of your own. This could be a large variety of decentralized applications, such as secure chat communcation, file-sharing, or efficient mesh networking.

CHAPTER 2

Advanced Features

This section outlines advanced features that are or will be available in *Cicada* in the official 1.0 release.

2.1 Attacker Resilience

Traditionally, if a peer were to communicate with another peer, the traffic would take a single route through the network topology to get to the other peer. If there is a malicious agent in the network, they could rewrite unencrypted traffic and inject arbitrary payloads. To work around this, traffic can be forked and sent through multiple peers throughout the network simultaneously. This increases overall load on the network, naturally, but is a small price to pay in ensuring that your data isn't messed with in-transit.

To use this feature, pass the `duplicates` keyword argument on a *per-message* basis when using the API:

```
peer = SwarmPeer("localhost", 10000)
peer.connect("10.0.0.1", 50000)
peer.send(("10.0.0.2", 50000), "hello!", duplicates=5)
```

2.2 Visualization

When running the *Cicada* visualization tool, `visualizer.py`, there are a number of controls for manipulating the behavior of the peers:

- Press **R** to join all peers together into a single network at random.
- Click a peer, press **J**, then click another peer in order to join the former to the latter.
- Pressing **L** between peers performs a lookup on the network on the latter's ID.
- Select a peer and press **F** to dump the peer's finger table.
- Select a peer and press **P** to dump its full list of known peers.
- Select a peer and press **B** to send a broadcast packet to the entire network that peer is connected to.

CHAPTER 3

Feature Work

There is still a long way to go before *Cicada* has a robust enough feature set for general consumption; this section outlines future plans.

3.1 Port Forwarding

Most people use devices on personal networks, and are thus hidden behind a router that is doing **n**etwork **a**ddress **t**ranslation (NAT). Similar to how BitTorrent needs to temporarily open ports in order to seed content, we need to do likewise in order to facilitate new peers into the swarm through a local peer. To do this, we use similar techniques to libtorrent, namely NatPMP and UPnP. These will allow you to create a swarm peer without worrying about whether or not it will be able to be accessed from the Internet. **Estimated Release**: 0.3.0-alpha

3.2 Security & Encryption

In a peer-to-peer network, it's impossible to determine what peers your traffic will travel through on the way to its destination. Standard routing through the Internet faces these same implications, but we implicitly trust that network topology more (we must, in fact, in order to gain any semblance of security).

The only way to *ensure* secure communications that are immune to Man-in-the-Middle attacks and packet sniffing is to establish a trusted set of encryption keys before using the network. This can be via secure email, and encrypted telephone call, exchanging symmetric keys in person, etc. Once these keys are exchanged, *Cicada* can use them directly to encrypt all outgoing communication to a particular peer.

If you trust the network (or at least the majority of it – see the *Attacker Resilience* section), you can use standard public-key authentication methods to establish an SSL communcation stream between particular peers. That is to say, the traffic is still routed through the other peers, but is encrypted with SSL. **Estimated Release**: 1.0.0-rc

If you want to hard-code secret keys, configure a key file like so (choosing one of either `"peer"`, specifying the exact peer ID, or `"address"`, specifying the `host:port` pair of the peer):

```
{
  "trusted_hosts": [{
    "peer":
↪"24355304810235874286134060455083535315455785472150272366747243996307578662525",
    "address": "75.23.66.101:7000",
    "outbound_key": "outbound_encryption_key",
    "inbound_key": "  inbound_encryption_key"
  }, {
  }]
}
```

Then, just pass it to the command-line. Any communications between the localhost and the peer at `75.23.66.101:7000` will be encrypted *if the other peer is also aware of the encryption keys*:

```
$ ./cicada.py -p 7001 --join 75.23.66.101:7000 --keys keylist.json
```

3.2.1 API Documentation

Here we outline detailed usage of the *Cicada* API; it explains much more than the *technical* details under, say, `help(swarmlib.swarmnode)`. For examples and tutorials, see the *Tutorials* page.

Interacting with *Cicada*: `SwarmPeer`

This object is the main way of interacting with the *Cicada* API as it wraps the lower-level DHT and routing details. It mimics the official `socket.socket` interface as closely as possible.

Unlike that interface, though, a `bind` is *required*, since every peer in the swarm acts like a server for all the others.

class swarmlib.**SwarmPeer**([*hooks={}*])

This creates an object, registering a selection of callbacks that hook into various lower-level functionality. The valid keys into `hooks` are:

- `"send"`: called for every sent packet. it's called with the following signature: `send(PeerSocket, bytes)`, where the `PeerSocket` parameter is responsible for sending the data. this includes *all* messages, including the ones that occur at a lower level, such as the DHT layer.
- `"recv"`: called for every time a full high-level data packet is received. it's called with the following signature: `recv(RemoteNode, bytes)`, where the `RemoteNode` parameter is the node that the full data packet was received from.
- `"new_peer"`: called for every time a new *SwarmPeer* joins the swarm: `new_peer(PeerSocket)`.

SwarmPeer.**bind**(*addr*, *port*[, *external_ip=None*, *external_port=None*])

Binds to a particular address, establishing the listener for this member of a *Cicada* swarm. A subsequent *connect()* indicates a peer joining an existing swarm, whereas a lack there-of indicates a peer establishing itself as the first member of a swarm. The optional parameters (which must *both* be specified) set a custom ID for the peer. **This is the first method that must be called on a *SwarmPeer* instance, before any packet operations.**

Parameters

- **addr** (*str*) – either the IP address of a local interface (such as `eth0`), a hostname like `localhost`, or an empty string, which would indicate a binding on *all* interfaces.
- **port** (*int*) – the port to bind on, in the range [1025, 65535)

- **external_ip** (*int or None*) – the external IP address of your host on the network. this applies to NAT traversal situations as seen in the *example below* where you don't immediately have access to your external network or a port forwarded on your router. see the *traversal* module for details.
- **external_port** – as with the IP, this is the mapped external port.

```
from cicada import swarmlib, traversal

peer = swarmlib.SwarmPeer()
with traversal.PortMapping(5000) as pm:
    eip = pm.mapper.external_ip
    peer.bind(pm.local_address, pm.port, eip, pm.eport)
```

SwarmPeer.**connect**(*network_host*, *network_port*[, *timeout=10*])

Connects to a peer in an existing swarm.

Parameters

- **network_host** (*str*) – the IP address or FQDN of an existing *Cicada* swarm.
- **network_port** (*int*) – similarly, the port of the listening peer
- **timeout** (*int*) – after this amount of time (in seconds), the call will immediately return.

SwarmPeer.**broadcast**(*data*[, *visited=[]*])

Broadcasts data to the entire swarm. For details on the broadcasting algorithm, you can read this blog post.

Parameters

- **data** (*bytes*) – the raw data to sendtarget (tuple) – one of the following: a 2-tuple (hostname, port); a Hash; or another SwarmPeer instance
- **visited** (*list*) – this parameter is largely used internally to the SwarmPeer object to perform efficient broadcasting, but can be otherwise specified by the caller in order to indicate the specific peers that should be excluded from the broadcast. the list should contain Hash objects.

SwarmPeer.**send**(*target*, *data*[, *duplicates=0*])

Sends a data packet into the Cicada network.

Parameters

- **target** (*tuple*) – one of the following: a 2-tuple (hostname, port); a *Hash*; or another SwarmPeer instance
- **data** (*bytes*) – the raw data to pack and send
- **duplicates** (*int*) – the amount of extra peers to route the message through; this is related to *attacker resilience*.

SwarmPeer.**recv**()

Blocks until a data message is received from the Cicada network.

Return type (SwarmPeer, bytes, bool)

Returns the source peer that the message came from, the data message we received, and whether or not there are more messages pending

Developer Note

This actually returns `RemoteNode` instance rather than a `SwarmPeer`, currently, because I haven't finished implementing that yet.

NAT Traversal Methods

See the *NAT Traversal* tutorial for examples.

class `traversal.`**PortMapping**(*port*[, *protocol="tcp"*])

Establishes an external port mapping using the NAT traversal methods: UPnP, then NAT-PMP. It's intended to be used using Python's `with` construct. See *this example* for a use-case.

If you wish to use one of the port mapping modules specifically, see the documentation for the `UPnP` or `NatPMP` objects.

Parameters **port** (*int*) – this is the *requested* port to perform an external mapping on. if the port is already mapped, the `with` clause will exit immediately; see the *eport* attribute for the resulting port mapping.

`PortMapping.`**eport**

Specifies the external port that the mapping succeeded on; this may or may not be the initial port that was passed in.

Low-Level Interaction

Routing

These objects are used in various places to coordinate routing in the *Cicada* network, such as specifying a send target (instead of a raw address tuple).

class `chordlib.routing.`**Hash**([*value=""*, *hashed=""*])

Either you know the initial value and the hash is computed, or you know the hashed value (and its initial value is – by definition – not determinable) and only that is stored.

Custom Swarm Creation

Maintainer's Note

The documentation in this area is much less frequently maintained, as its not intended for consumption. It's merely a starting point for anyone that isn't really interested in *Cicada* and more interested in creating their own DHTs.

This section outlines methods for creating custom swarms by interacting directly with the raw distributed hash table (DHT) objects. All of the objects outlined here *cannot join or otherwise interact with a Cicada swarm*, unless they understand the higher-level protocol's expectations.

class `chordlib.localnode.`**LocalNode**(*data*, *bind_addr*[, *hooks={}*])

Creates an unconnected peer in a Chord DHT.

3.2.2 Runtime Interpreter

By passing a file to `cicada.py`, you can execute a series of commands that interact with the *Cicada* API.

Each line is executed in order, and has a set of required parameters. Comment lines start with a # and are ignored.

- `SEND [host] [port] [data...]` Sends a message to a particular address. `[port]` must be convertible to an integer.
- `RECV [count]` Waits for a message to be received from anyone. The `[count]` parameter is optional, and indicates the number of messages to wait for.
- `BCAST [data...]` Sends the specified data to the entire swarm.
- `OPT key1=value key2=value [key=value...]` Processes and sets the *configurable options*. All subsequent lines will have these options applied to them.
- `WAIT [time (s)]` Waits for an incoming connection for a certain amount of time. If set to -1, waits indefinitely. This is useful for the first peer in a swarm.

Configurable Options

Parameter	Type	Description
duplicates	`int`	Configures the number of extra paths to take for all of the `SEND` calls.

Example Runtimes

You usually will want a single "server" runtime that starts the swarm; it waits for the other peers to join. This can look something like this:

```
WAIT -1
OPT duplicates=3
BCAST Hey everyone, I'm the original peer.
SEND localhost 49611 Hey there, specific client @ localhost:49611, it's me.
RECV 2
```

You would run this like so:

```
./cicada.py --interface localhost -p 49610 --no-port-mapping first.conf
```

Similarly, you'd want a "join immediately" peers that look something like this:

```
RECV 1
BCAST Hey everyone, I'm a new peer!
RECV 1
SEND localhost 49610 Hey there, localhost:49610; it's me.
```

This would be run like so:

```
./cicada.py --interface localhost -p 49611 --no-port-mapping --join localhost:49611␣
↪others.conf
```

3.2.3 Tutorials

Jump to a Tutorial

- *Tutorials*

NAT Traversal

This is used for peers behind a router, as is the case for most users. There are two methods of external port mapping, and both are covered here. To see usage of "catch-all" port mapping, just reference the *example* in the API docs.

Note

This example should not be necessary in a few iterations of the library. It's necessary now because detecting whether or not the peer is behind a router (and thus needs NAT traversal) is not implemented. Eventually, this part will happen automatically.

UPnP

Universal **P**lug a**n**d **P**lay is the first method used when using the generic *traversal.PortMapping* object.

The following example tries to map the local port 7777 to external port 8888, increasing the mapped port up to 5 times on failures.

```
import sys
from cicada.traversal import upnp

LOCAL_PORT = 7777
ATTEMPTS = 5

mapper = upnp.UPnP()
mapper.create()

eport = 8888
for i in xrange(ATTEMPTS):
    if mapper.add_port_mapping(LOCAL_PORT, eport, protocol="udp"):
        break

    if i == ATTEMPTS - 1: continue
    print "Failed to map %d <--> %d, trying %d." % (
          LOCAL_PORT, eport, eport + 1)
    eport += 1

else:
    print "Failed to map %d after 5 attempts." % LOCAL_PORT
    sys.exit(1)

print "Succeeded in mapping %d <--> %d." % (LOCAL_PORT, eport)
mapper.delete_port_mapping(LOCAL_PORT, protocol="udp")
mapper.cleanup()    # removes *all* mappings
```

NAT-PMP

This method is often used on Apple routers and is the backup method tried after *UPnP*. Using this method is actually *identical* to *UPnP*, as the API is designed to be identical. The only difference is that you should use the `traversal.NatPMP` instance instead.

Simple 2-Node Echo Server

In this sample, we'll create a 2-peer *Cicada* swarm and echo messages back and forth between the peers.

```
""" Establishes a *local* swarm, echoing a message between them.
"""
import sys, time
from cicada import swarmlib
from cicada.traversal import portmapper

local_address = portmapper.PortMapper.get_local_address()
first, second = swarmlib.SwarmPeer(), swarmlib.SwarmPeer()

first.bind(local_address, 5000)
second.bind(local_address, 5001)
second.connect(*first.listener)

first.send(second.listener, "hello!")
src, data, _ = second.recv()
assert data == "hello!"
assert src.chord_addr == first.listener

second.send(first.listener, data[::-1])
src, data, _ = first.recv()
assert data == "!olleh"
assert src.chord_addr == second.listener
```

Python Module Index

Index