Lab 4: Pd and Python Communication

In this lab, we will be discussing how to communicate back and forth between Pure Data and Python. A number of patches and scripts have been prepared and shared with you. Feel free to use these patches/scripts in your projects as well (if applicable)

Overview

We will be going through 4 examples for each of which a python script and a pure data patch has been provided. These examples are:

1. Sending OSC messages from python to pd
2. Sending OSC messages from pd to python
3. Two-way communication between python and pd (synchronized with a clock signal sent from pd)
4. Two-way communication between python and pd (synchronized using timestamps and scheduler)

Preparing Environments

For the pure data patches, no external dependencies are required. However, for running the python scripts, the following packages are required:

1. Python-osc:

   https://pypi.org/project/python-osc/1.7.4/#description

   ```
   pip install python-osc
   ```

2. APScheduler

   https://apscheduler.readthedocs.io/en/stable/

   ```
   pip install apscheduler
   ```

Exercise 1: Python to Pd

In this exercise, we use python to generate random notes at given time intervals. The generated notes will have the following characteristics:

1. Pitch → midi number sampled from a range within (0 - 127)
2. Velocity → an integer sampled from a range within (0 - 127)
3. Duration → length of a note in ms sampled from a given range

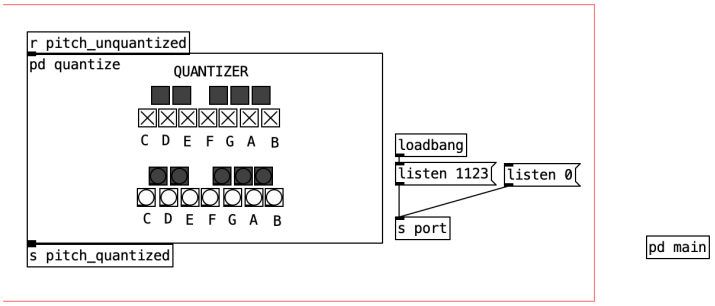
The generation is done using a class implementation called NoteGenerator

```
class NoteGenerator:
    def __init__(self, min_pitch=48, pitch_semitone_range=24, vel_range=(0, 127), dur_range=(10, 1000)):...

    def generate(self, n_examples=1):...
```

The connection with pd is established using OscSender and messages can be sent using the OscSender.send_to_pd() method

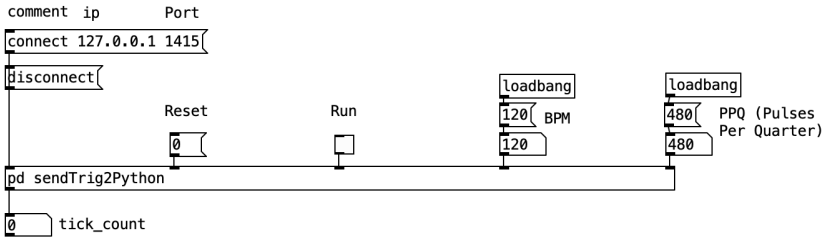
```
class OscSender:
    """
    Class for sending OSC messages from python to pd
    This class establishes a connection with pd server on ip address and sending_to_port
    """
    def __init__(self, ip, sending_to_port):...

    def send_to_pd(self, message_parameters, message_values):...

    def get_ip(self):...

    def get_sending_to_port(self):...

    def get_client(self):...

    def change_ip_port(self, ip, port):...
```

The provided pd patch will receive sent messages from python by establishing a UDP connection and decoding the messages via the oscparse method



Exercise 2: Pd to Python

The following pd patch is used to send messages to python



The patch here sends the tick position periodically at specific times. The duration between each two sent messages is 60000/(BPM*PPQ) ms. This message can be thought of as a clock message syncing python to pd (more on this in Exercise 3). The sent message is formatted as

```
/clock/tick Tick_Position
```

On the receiving end, python uses the OscReceiver method to receive the messages. The received messages are then handled according to the defined handlers for the messages. In this exercise, python only receives the clock position and prints it out to the console.

```python
class OscReceiver(threading.Thread):

    def __init__(self, ip, receive_from_port, quit_event, address_list=["/clock*"], address_handler_list=[None]):...

    def run(self):
        # When you start() an instance of the class, this method starts running
        print("running --- waiting for data")

        # Counter for the number messages are received
        count = 0

        # Keep waiting of osc messages (unless the you've quit the receiver)
        while not self.quit_event.is_set():

            # handle_request() waits until a new message is received
            # Messages are buffered! so if each loop takes a long time, messages will be stacked in the buffer
            # uncomment the sleep(1) line to see the impact of processing time
            self.server.handle_request()
            count = (count+1)                    # Increase counter
            print("count {}".format(count))      # Print current count, i.e. total number of messages received
            #time.sleep(1)
```

Here we introduce the concept of concurrency. As opposed to Exercise 1, the receiver class is a child class of the threading class.Thread class. We'll discuss this further in the lab, but for more information, you can refer to :

[1]https://realpython.com/intro-to-python-threading/
[2]https://www.bogotobogo.com/python/Multithread/python_multithreading_subclassing_creating_threads.php

Exercise 3: Two-way communication

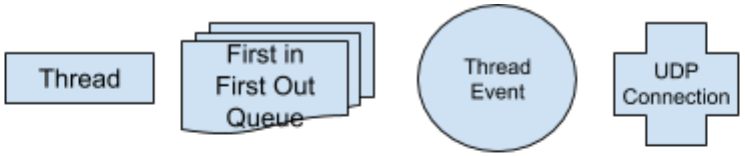
Synchronized with a clock signal sent from pd

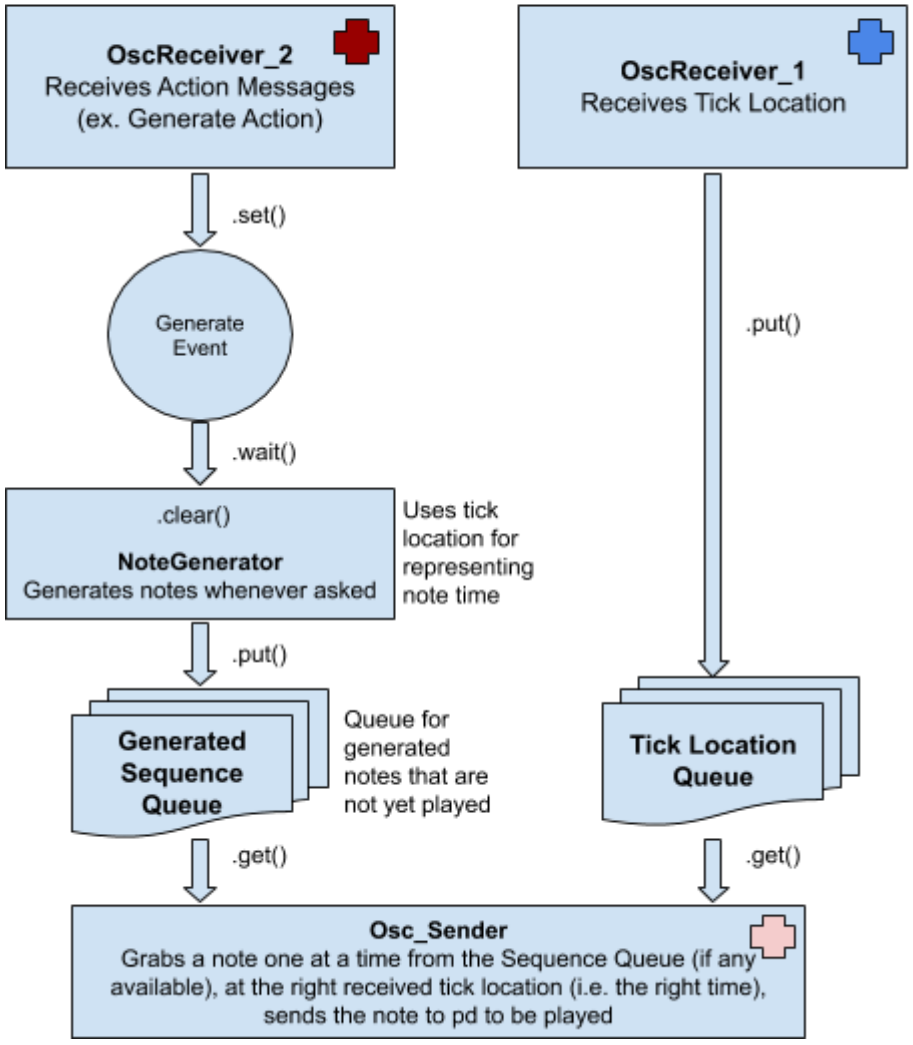
In this exercise, we implement two-way communication between python and pd. The implementation roughly follows the scheme provided on the next page.

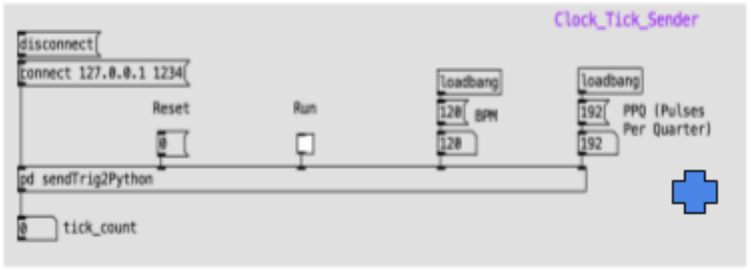
For this exercise, all of the processes in python are running as separate threads. This means that a process is active ONLY when a message is received or when a resource is available. For example, the note generator can carry out the generation process while the OscSender can send a note in parallel. Moreover, while the generation and transmission of notes are being carried out, a separate thread can receive the tick location (sync signal) from pd.

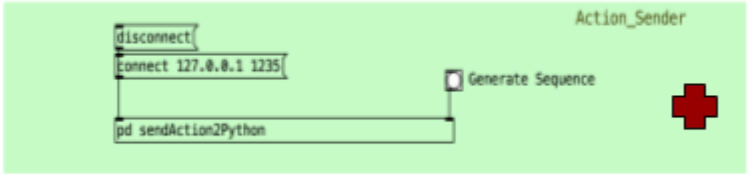
The sync signal is used to synchronize the playback of the notes in python with pd. In this exercise, the note generator generates notes based on the tick location on which they should be played. Likewise, the OscSender constantly checks whether a note should be played based on the tick position received from pd.

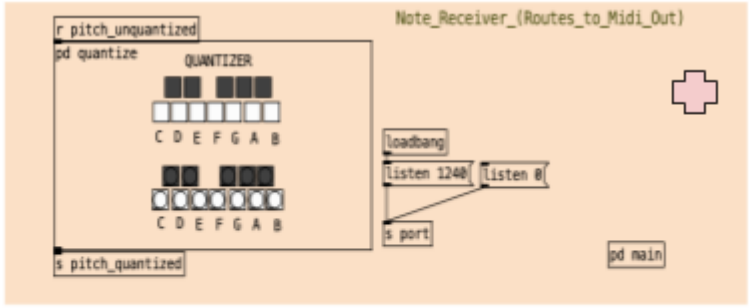
For shared resources, python Queues are used. Queues are sort of similar to stacks except that they work on a first-in-first-out basis.


Thread
First in
First Out
Queue
Thread
Event
UDP
Connection


OscReceiver_2
Receives Action Messages
(ex. Generate Action)
.set()
Generate
Event
.wait()
.clear()
NoteGenerator
Generates notes whenever asked
Uses tick
location for
representing
note time
.put()
Generated
Sequence
Queue
Queue for
generated
notes that are
not yet played
.get()
OscReceiver_1
Receives Tick Location
.put()
Tick Location
Queue
.get()
Osc_Sender
Grabs a note one at a time from the Sequence Queue (if any
available), at the right received tick location (i.e. the right time),
sends the note to pd to be played


Clock_Tick_Sender
disconnect
connect 127.0.0.1 1234
Reset
Run
loadbang
120
BPM
loadbang
192
PPQ (Pulses
Per Quarter)
pd sendTrig2Python
tick_count


Action_Sender
disconnect
connect 127.0.0.1 1235
Generate Sequence
pd sendAction2Python


Note_Receiver_(Routes_to_Midi_Out)
r pitch_unquantized
pd quantize
QUANTIZER
C D E F G A B
C D E F G A B
loadbang
listen 1240
listen 0
s port
pd main
s pitch_quantized

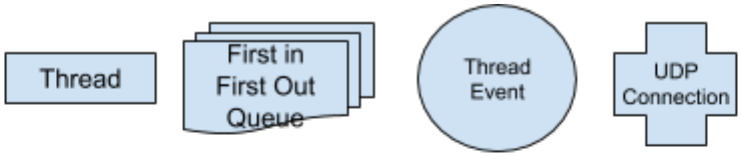
Exercise 4: Two-way communication

Synchronized with a clock signal sent from pd

In this exercise, we modify the previous implementation such that the playback is not synchronized with the clock signal received from pd. Rather, we will use APScheduler to schedule the playback of events based on the exact timing information. In other words, in this case, the note generator generates notes while specifying the exact time that they should be played on.

A scheduler is basically a utility that allows for methods/processes to carry out either repeatedly or at specific designated times.

See the diagram below for an overview of the implementation.



OscReceiver_2
Receives Action Messages
(ex. Generate Action)

.set()

Generate
Event

.wait()

.clear()

NoteGenerator
Generates notes whenever asked

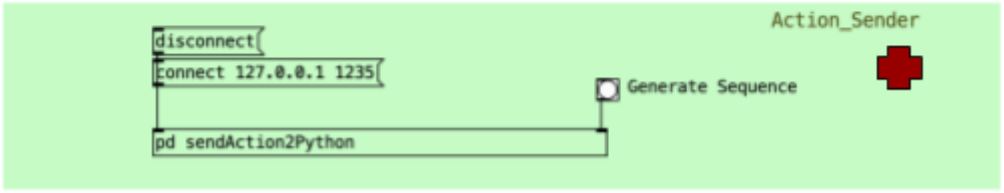
Uses absolute time for representing note time

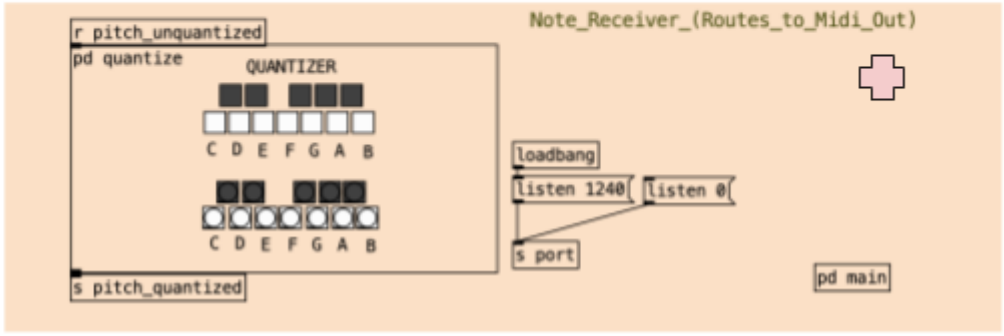
.put()

Generated Sequence Queue

Queue for generated notes that are not yet played

.get()





NoteSequenceQueueToPlaybackScheduler

Whenever a new note is available, it grabs the note and schedules an OscSender object to send the note to pd at the exact time as the note should be played

Tasks for Lab 4:

Using the code provided in Exercise 4:

1. Make sure the provided patches work on your personal computer [1 Mark]
2. Modify the note generator such that the generated note durations are multiples of a 16th note (rather than random length durations) - i.e. the timings should be quantized to a 16th note resolution [3 Marks]
3. Add random micro timing variations to the onsets of the generated notes in (2). The microtiming should be within a ±32nd note resolution. [3 Marks]
4. Update the generation such that the generated notes can only belong to certain pitch classes (i.e. implement a pitch quantizer in python). [3 Marks]

BONUS: Allow for the pd patch to dynamically modify BPM and PPQ parameters in the python program.

Submission:

- Your pd patch
- The python scripts
- A short video for each part functioning (preferably with a screen recording software, or alternatively with your phone). For each video, there should be a complementary document describing what has been the approach, whether the code is working, and if not, what may be the source of the issue. Alternatively, you can use narration over the video recordings to address these questions (in this case, you don't need to submit an additional written document.)