Introduction to Real Time Computing Lab Report

Akshit Rawat SIGMA M1

August 2024

1 Abstract

The aim of the experiment is plotting streamed data using a recursive data generation function on a Banyan backplane, which would mimic a data stream coming from the Arduino Board when used for data acquisition. It is also to show the event driven programming using subscriber-publisher protocol, the architecture of which would be based on Python Banyan framework, between server and client, where server would be generating data and saving it in local buffer, and then send the stored data to client's buffer for later plotting which would refresh every 0.5 seconds.

2 Introduction

Asynchronous event driven programming is a type of programming in which multiple processes can be triggered using events. This means one process does not have to wait for the other process to finish off, which generally happens in sequential programming. We can implement this style of programming using a messaging protocol. One of the messaging protocol which we have used here is the framework called Python Banyan. It is a framework designed using object oriented programming where new child classes created to establish protocol between publisher and subscriber components inherit functions and attributes from the Banyan's base class. The Banyan framework uses the publisher-subscriber protocol which sends messages from transmitter to receiver based on Sender-Message-Channel-Receiver (SMCR) approach. Sender would be the one who publishes the message and sends through the Channel, ready to be received by the Receiver who would be the "Subscriber". The network traffic, subscriber and publisher ports are all handled by the banyan socket called the Backplane, which would be the "Channel" in the SMCR approach.

The Backplane receives all the messages and give it to destination based on the subscriber port,topic and payload. The topics are of type string and are attached to the payload being published for the subscribers. It is what creates the events, the sending and receiving of payload with topic, in response of which the processes initiate or change. A message payload is what carries the required information and the reception of this payload is the event that triggers further action on the receiver side. The information is packaged in the form of dictionary, with keys and its items. The topics are attached to the message payload so that the specific payload can be sent to evoke responses cause by specific topic.

To break down the processes of whole messaging protocol further, we start with the sender. The sender has the information source. It has the transmission mechanism in which it will

package the information into a messaging payload with topic and encode it to make it ready to be sent to the receiver end. The protocol for packaging and encoding (done by MessagePack) is done when the payload is published. It is done using zeromq protocol which is part of the banyan base. The message is then published to be sent to the receiver through the backplane, and would receive all messages and alert about incoming new messages to the connected receiver/receivers. Next, the receiver has the receiving mechanism, which is the event loop, and message processing function, the event handler, with it. When it receives the payload, it first will match the topic string with its subscriber topic, if it matches then it is put on the receive queue which after that goes through the mechanism of decoding so that the message can be again seen as a dictionary, it is then passed down to message processing function for further exploitation. Decoding is carried out using the zeromq protocol, which uses MessagePack, in the receiving mechanism.

For our experiment we establish this protocol between a server and a client. Here, both components act as subscribers and publishers. The model function is to show event driven functionality governed by the exchange of topics between server and client, which carry out event driven process, that is the data generation and plotting the data streamed in the client interface sent over by the server. Brief explanation is as follows. First, the server and client do a handshake mechanism with the help exchanging initial connection topics to make sure that both are connected to the Backplane and ready in the receiving ends, this way both notify to each other. Now, coming to the later functioning, the component server function is to constantly generate new samples and put in its local buffer. The buffer would collect and form a list of samples, which are in the form of tuples, and then put inside the payload in the form of dictionary with a topic. The topic has to be same as the topic set by the subscriber, which is set by client. When the client receives the payload it would first decode it so that it can be read in the form of dictionary again and then an event triggers the action, because of which it is passed into a buffer for storing, so that it can be read for further plotting. This is the response which gets triggered by receiving the topic 'plotting'. Once a batch of list has been plotted, it asks for more data by sending a message payload with topic "request". The receiving loop receives the topic, decodes it and sends the message to the function to be executed under the same topic. Once the topic is received, the server which is ready with a batch of data, sends it again. Once the client has reached its plotting limit, it would send a topic called 'interrupt', meanwhile when the server listening for the topic, receives it, it would stop the generation and the plotting would end. The detailed explanation with code has been provided in next section.

3 Model and Design choices

The Server and Client design is based on the Banyan base class which it inherits using the python built in function called super. It gives access to the method and attributes created in the base class. Methods can then be overridden and adjusted according to the design choices made by us, because of polymorphism. The methods/functions such as set-subscriber-topic and publish-payload and receive loop can be called inside the child classes by the use of self prefix, as the object inherits all the functionality defined in the base class. The encapsulation, as practised in this model, binds all the methods and attributes inside the class and keeping them secure. Because of super function we can manipulate base attributes which would now be part of the child classes. Base attributes such as loop time, receive-loop-idle-addition can now be manipulated based on our preferences.Lastly, the Backplane has to be run first otherwise there would be no socket to connect the server and clients to, no channel to send messages or objects.

Backplane is the source which establishes the subscriber ports and addresses so that the

server and client do not have to deal with the addresses to which they are sending the data. Backplane would have the pool of messages which would later be collected by the client and server based on the topics. The client and server only have to rely on topics to drive their specific event driven processes. The first and foremost thing to establish publisher-subscriber protocol design is the handshaking mechanism in which the client send the server that it is connected to the server by sending a message to it. Then, the server also send the client that it is up and running and connected to the backplane.

3.1 Server Side

3.1.1 Initialisation Method

```
class BuffServer(BanyanBase):


    def __init__(self):

        super().__init__(process_name = 'BuffServer',
        receive_loop_idle_addition=self.datageneration_loop,loop_time = 0.1)
        self.set_subscriber_topic('request')
        self.set_subscriber_topic('interrupt')
        self.set_subscriber_topic('connected')
        self.server_fifo = FIFO(2)
        self.t = 0
        self.xt = 0

        self.topic_=False
        self.topicc=[]


        try:
            self.receive_loop()

        except KeyboardInterrupt:
            self.clean_up()
            sys.exit()
```

The class BuffServer is the Server object for which it inherits the methods, attributes and functionalities of the base class called BanyanBase. The Super function is defined inside the child class constructor method called –init–. This is done, so that the super function can call the –init– method of the base class. This way the inherited attributes get initialised in the child class.

The instance attributes created are such that they get initialised. self.t=0 and self.xt =0 are the initial values which would be passed on to the generate-next-sample function, which is inside the data generation loop and these would be its starting values (initial samples). The self.topic-=False has been created because it would be used as a boolean flag to control the state of the data generation. It is readily false so that no data generation start as soon as server

is on. It is triggered to True once the server receives the topic='request' and return to false again when the server receives the topic = 'interrupt'.

It is important to initialise the subscriber topics and receive loop as they are the function that would run first and also would establish the publisher subscriber protocol. set-subscriber-topic establishes the topic to which the client would publish its payload to create the event. Initial connection between the server and client is established using the subscriber topic = 'connected', once the server receives it, it ensures that the client is connected. Then it sends a message with topic 'me too' to the client to make the client ensure that it is also connected to the backplane. After this the server starts receiving the 'request' topics so that it can send the data to the client.

The receive loop carries out the decoding of the message payload so that it can be accessed as a dictionary and later be processed by the server. It first carries out the zmq protocol of receiving the published envelope and then unpacks the data. The zmq protocol, because of which the receiving loop listens for messages without stopping is because of zmq.NOBLOCK flag. It makes the process non blocking. If the no message is received, it throws an exception which executes the idle addition loop which is used for running extra methods in mean time. After reception of payload, the receive loop sends the decoded, unpacked data to the event handler called incoming-message-processing function to carry out further processing. In this, the further processing would be to trigger the data generation loop and publishing a message payload containing the samples to be sent to the plotting client.

Keyboard Interrupt has been introduced so that the server can be interrupt abruptly. Lastly, the FIFO buffer has been instantiated as the attribute of the class Server. Because of this, we are able to use the methods defined in the FIFO Buffer. The attributes of the FIFO buffer are private so that they cannot be manipulated. This aspect of encapsulation allows only specific methods to gain access to the FIFO buffer, which have been defined in the class. It keeps the FIFO secure and maintain its prime behaviour or state. The FIFO buffer has been set to have temporal window of 1s. This is done as the refresh time of plotting client is set to be 0.5s and the sampling period and loop time of the data generation loop is set to be 0.1s. Therefore, the number points generated would be approximately 5 and according to the design of buffer, the buffer expands until the current time being stored inside the buffer is not bigger than the temporal window which is of 1s. Therefore, it would store up to 10 data points, meaning from 0.0 to 0.9 and would not overwrite until it gets 1.1s. Loop time of data generation loop , like 0.01, would result in overflow/overwriting in buffer as the buffer can only hold 10 data points, and with faster loop time, there would be more data point generation. Therefore, this configuration is sufficient for this sample period and refresh time.

3.1.2 Message processing function

```
def incoming_message_processing(self, topic, payload):

    if topic=='connected':


        print("client connected")
        self.publish_payload({'loop_time':0.5},"me too")
```

```
if topic=='request':
    self.topic_ = True
    self.topicc.append(topic)
    print(self.topicc)


    cb_data = self.server_fifo.read()


    print(cb_data)
    if not cb_data:
        return
    else:
        payload = {'data':cb_data,'time':self.t}
        self.publish_payload(payload,"plotting")
        self.server_fifo.clear()
        self.topicc.clear()


elif topic == 'interrupt':
    self.topic_ = False
    print("client interrupt")
    self.server_fifo.clear()
    input("Press Enter to exit")
    self.clean_up()
    sys.exit(0)
```

Incoming-message-processing function is the even handler, which initiates when the receive loop is ready with the decoded message/object, which then gets passed into this function. The function carries out three events, which are topic ='connected', topic='request' and topic='interrupt'. The server first receives the topic 'connected' which notifies the server that the client is now connected to the Backplane. After knowing, it sends the notification with topic 'me too', which notifies to the client that the server is up and running and connected to the backplane, ready to send the data. After this, the servers starts receiving the topic 'request' for sending the data. When the server receives the 'request', the self.topic- flag becomes True and triggers initiates the data generation. This also triggers the reading of the local FIFO buffer, so that stored samples can be sent. The event, reception of the topic 'request' drives the further processes of generating, reading and sending data. Inside, there is another if condition which has been created due to the fact that in the initial moments of receiving the topic 'request', buffer would be empty and it would send a payload with empty list, which would cause an error in the client side as an empty list is not valid to be stored in a Circular buffer. This is how the server also gets notified that the client is connected to the Backplane.

So, the next time it receives the 'request', the data would in the buffer, ready to be read and published. Next, we use publish-payload with payload containing the samples and topic set to 'plotting'. After publishing, the buffer is cleared so that new data can be stored simultaneously.

Adding on, the Topic collector list (topicc) has been created to check what events have occured based on topic names. When the topic = 'interrupt', the topic- flag changes to False, and then it changes the condition of the data generation loop by moving it to an else condition returning Boolean flag status as False. Buffer is also cleared and then the exit function is used so that all the processes get stopped and server exits.

3.1.3 Data Generation Functions

```
    def datageneration_loop(self):


            if self.topic_:
                (self.t,self.xt) =self.generate_next_sample(
            current_sample=(self.t,self.xt),tau=0.1,sigma=1)

                self.server_fifo.write(
                (self.t,self.xt), current_time=self.t)
                print(self.topic_)

            else:
                return print(self.topic_)









    def generate_next_sample(self,current_sample = None, tau=.01,sigma=1):
        if current_sample is None:
            return 0, 0
        else:
            T = invgauss(mu=tau)
            dt = T.rvs()
            X = norm(loc=0, scale=sigma)
            t, xt = current_sample
            t += dt
            xt += X.rvs() * sqrt(dt)
            return t, xt

def buff_server():
    BuffServer()


if __name__ == '__main__':
    buff_server()
```

Here, I would first address the bottom lines of the code. They are the ones which when read, runs the class and then its functions. The if condition checks whether the name, which is

an inbuilt variable of every python module which changes to –main– when ran, becomes equal or not. If it is true, then execute the function containing the server class.

Now, the function data generation-loop is an idle addition loop which runs when there is no event happening. It runs when receive loop is running idle and not receiving any messages. The idle loop time is set to 0.1 seconds as this will also be the sample generation buffer writing speed. In the data generation loop if condition calls the sample generator method to generate new sample every time it runs and it has the FIFO buffer writing function. The if condition gets triggered when the topic- flag turns true. After turning on, the sample generator generates tuples containing time stamps and samples with varying values.

The sampling period is tau, time constant, which is set to 0.1 and sigma is set to 1 which is the standard deviation. Tau is the mean for the inverse Gaussian distribution object instantiated as T, which is then used to produce successive dt values which would vary around 0.1 using function rvs(), randomly picks the number from the T. The variable can go upto 0.2 and minimum can go up to 0.02,a rough estimation. The xt values are governed by the standard deviation 'sigma' of the normal distribution object with mean as 0. X is the instance of the normal distribution object. The difference between xt values would be from -1 to 1, therefore higher sigma values will show higher fluctuations in the xt values. The sample period had to be equal to idle loop time so that the data generation is in sync with the amount of data produced and be held by the buffer so that it does not overflows/overwrite.

Once triggered, the data generation would continue until the interrupt event occurs.

3.2 Client Side

3.2.1 Initialisation

```
plt.ion()
class BuffClient(BanyanBase):
    def __init__(self,refresh_time=0.5):

        super().__init__(process_name = 'BuffClient',
        loop_time=refresh_time,receive_loop_idle_addition=self.callloop)


        self.set_subscriber_topic('plotting')
        self.set_subscriber_topic('me too')
        self.refresh_time= refresh_time
        self.client_circ = CircBuff(size = 20)
        self.fig_circ, self.ax_circ = plt.subplots()

        self.publish_payload({'loop_time':self.refresh_time},'connected')
        try:
            self.receive_loop()

        except KeyboardInterrupt:
            self.clean_up()
            sys.exit()
```

The client object also inherits the same way as the server inherits the method and functionality from the Banyan Base class. Here, the loop time is equal to 0.5s, which is the refresh time required for updating the plot of data streaming. The receive-loop-addition we choose is the function formed called callloop. The callloop functionality is to send the request topic to the server so that it triggers the publishing of payload, and collection of new samples can be carried out. Firstly, the client will publish the payload with topic 'connected' to notify the server that is connected to the Backplane and it is ready to accept the data stream. The subscriber topic 'me too' is to accept the notification sent by the publisher which is the server. When the client receives this topic, it gets notified that the server is up and running and ready accept 'request' topic to send in the data. The subscriber topic 'plotting' is for whenever the publish payload is received with topic plotting, the client starts plotting by first writing the data in its local circular buffer and then reading it for plotting new samples.

The choice of buffer for plotting is chosen to be circular buffer. The buffer is configured to hold 20 samples. The size of the buffer corresponds to the seconds it will show every time on the screen. Therefore, it will show up to 2 seconds of data as the refresh time would be 0.5 seconds, therefore it will hold 5 data points for every 0.5 seconds. The circular buffer has been instantiated as the attribute so that it can be read and filled in anywhere inside the class. The circular buffer has the mechanism of filling in the slots in backward fashion, which saves it from unnecessary overwriting. Circular buffer are much faster in accepting data stream as compared to the FIFO linear buffers. Because of its fixed nature unlike FIFO linear buffers with varying size, the buffer does not require memory allocation like FIFO linear buffer.

The receive loop listens for the publisher payload with topic same as the subscribers topic. When the condition is met the message payload gets decoded using message pack function and then sends it to the incoming-message-processing function.

3.2.2 Message processing function

```
    def incoming_message_processing(self, topic, payload):


        if topic=='me too':
            print("server connected")
            self.publish_payload({'loop_time':self.refresh_time},'request')


        if topic == 'plotting':

            self.client_circ.write(payload["data"])

            print('update plot at t={}s'.format(payload["time"]))
            t_, x_ = zip(*self.client_circ.read())
            print(self.client_circ.read())
            self.ax_circ.clear()
```

```
            self.ax_circ.stem(t_, x_)
            self.ax_circ.set_xlim([payload["time"]-2,payload["time"]])
            self.ax_circ.set_ylim([-10, 10])
            self.fig_circ.canvas.draw()
            self.fig_circ.canvas.flush_events()

            if payload["time"]>=10:
                print("Plotting complete")
                self.publish_payload({'loop_time': self.refresh_time},
                'interrupt')
                input("Press Enter to exit")

                self.clean_up()
                sys.exit(0)


        if keyboard.is_pressed('l'):
            print("Interrupt sent")
            self.publish_payload({'loop_time': self.refresh_time},
            'interrupt')
            input("Press Enter to exit")
            self.clean_up()
            sys.exit(0)
```

The message processing function first accepts the topic 'me too' which is a notification event, notifying client that the server is well connected to the Backplane and up and running and ready to receive the data request. Therefore, it send the first request by publishing payload of topic 'request' to the server. The server receives and the data streaming begins with later "requests" carried out by the idle loop called the callloop method.

The 'plt.ion' which has been declared globally before creating Class is used to enable interactive mode in matplotlib, which allows for non-blocking updates to the plot. Because of this the figure or the canvas on which the data samples are being plotted would keep plotting as long as the event is being triggered. Because of this non-blocking nature, the receive loop, listening for the events can multi task by initiating idle addition loop and send request to trigger data generation and publishing payload.The object payload when accepted by the client, creates an event of accepting the message payload with topic "plotting". It is then passed to incoming message processing function for further processing. The event triggers a chain of actions when it checks that topic=='plotting'. First, it writes the received samples into the circular buffer. The circular buffer is read inside the zip function with zip(*). This means that the function unpacks the list of tuples and create two tuples out of the list. One tuple shows all the time stamps and the other shows the xt values. These are then sent for being plot. The stem function plots the data points in stem like manner where all the data points have a vertical and horizontal line. The base or the horizontal line connects all the data points from base.

The x-lim is the limit of showing plotted points on the screen at a specific time, here it would be every 0.5 seconds. The payload["time"] holds the value passed by the FIFO buffer in the server. The value gets updated whenever the current time parameter changes the value, as it holds the last value which is the highest in each list created by the FIFO buffer while writing. Therefore the plot updates with 0.5 seconds on the screen as the payload["time"] would hold

the last time stamp of published data samples. As every payload will have 0.5 seconds, the last time stamps of each payload will be spaced 0.5 seconds. So we will see updating plot every 0.5 seconds with 5 data points blasted on the screen every 0.5 seconds. canvas draw() and flush events () listen for the events and run sequentially once they receive the data on the canvas. The axis update depends on clear() and draw() and xlim function. The flush event () helps to process all the pending events, which are samples read from the buffer and put for plotting, immediately. It ensures that the plot figure window is up to date while plotting and in sync with the ever-changing buffer. The canvas draw() ensures that the update are reflected as desired on the screen.

The interactive plotting is event dependent. Therefore, it stops once the time stamp fulfills the if condition and then it exits with publishing a payload with topic = "interrupt" which launches an event to trigger the interruption of data generation loop. There is another mechanism through which I have introduced keyboard event. In this, when we press "L" on the keyboard, the plotting would stop and publish a payload which would initiate an interrupt event, resulting in interruption of data generation loop. This way we can stop the infinite data generation loop at any point or time stamp.

3.2.3 Request loop

```
    def callloop(self):
        self.publish_payload({'loop_time':self.refresh_time},'request')




def buff_client():
    BuffClient()


if __name__ == '__main__':
    buff_client()
```

The request loop is a receive idle addition loop and carries out the further loop of sending request topics to the server. It works when no transmission or reception events are taking place. In this design it plays a crucial role by continuing the event of sending topic "request" and data, which further continues triggers or production action of data generation loop in the server and further continuing to trigger reading of samples from servers FIFO buffer and then publishing payload. Right after, as the receive loop goes idle because no events are happening, the idle addition loop runs and sends request to the server such that the data stream remains continuous.

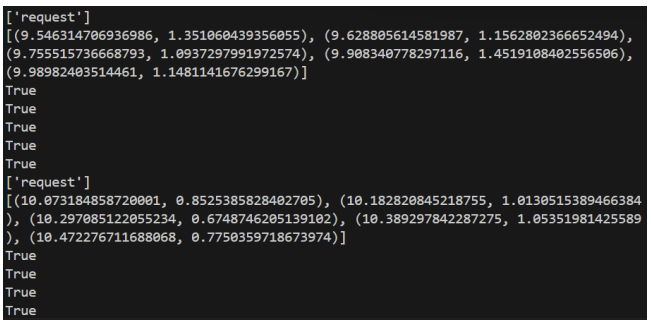

Figure 1: Event and Data transmission

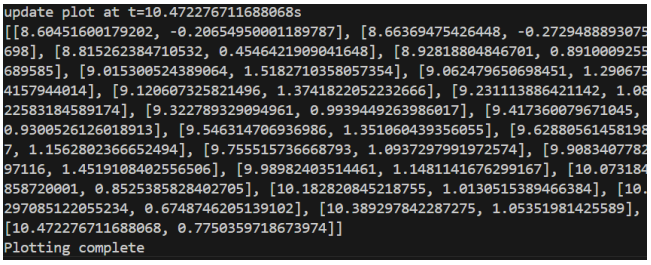

Figure 2: Data Reception

4 Observations

The observations are as follows. There were instances in the server side when the data generation loop produces six data points. It usually happens because of the high variability nature of the dt variable. The extra data-point produced has usually close time stamps with its preceding sample. Even though the six datapoints get produced before loop time, there was no loss of data because of the handshake mechanism which is initiated at the beginning. No loss of data is also insured by topic matching, continuous listening by event receive loop and message queuing and instant processing. This ensures that the payload reaches the destination and unchanged even if the loop times of both both server and client are different.

The sample period should match the loop time of the data generation otherwise the data generation would be slow or fast. If we go for lower values of the loop time with 0.1 as sample period, for example 0.01s, the data generation would be fast, which would lead to overflow as the FIFO buffer has limited size of accommodating 10 data points and has the functionality to overwrite data. It would mean that the data would already had gone through 10 cycles. This number of cycles would mean that the buffer has already overflown we would have skipped a lot of data points in between, which would lead to gaps in plotting figure as the refresh period is 0.5 seconds.

It was also important that there should be a trigger which would initiate the data generation loop only when the first event occurs, which is the reception of the payload with topic 'request'. This also ensures that the client receives data starting from initial t and xt values and does not receive later data points as soon as it connects.

To accommodate faster data streaming, which would mean short loop times, the buffer sizes would have to be configured and increased in order to maintain optimum data flow and also avoid unnecessary overwriting. To accommodate data streams with shorter sample periods, the circular buffer would be required to be reconfigured as it does not have a temporal window,

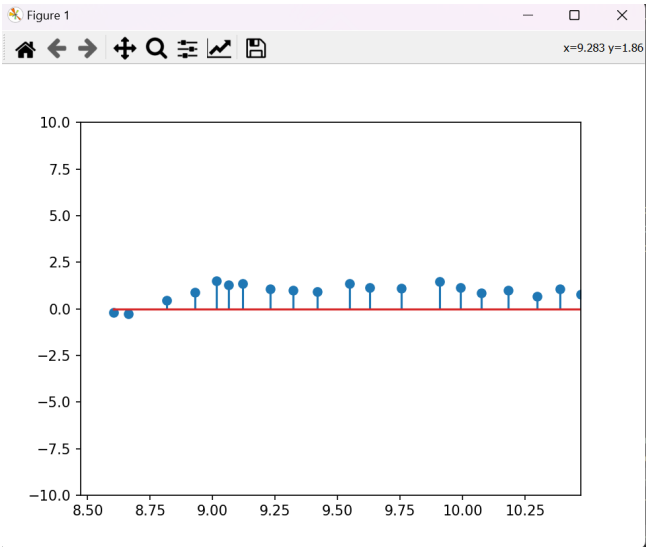

Figure 3: Plot representing last 2 seconds generated by final event

therefore to show 2 seconds of data with sample period of 0.01, the size should be 200 data points, otherwise the change would be seen in 0.1 seconds of plotting window. If the sample generation was kept inside the if condition of message processing function and buffer writing inside the data generation loop, the loop time of receive loop would not cause any effect on data streaming rate as then it would only get triggered by the event, which would happen every 0.5 seconds. Keeping 'plt.ion' outside of the Class maintains the global state of matplotlib to remain interactive at all times, ensuring that plotting remains event driven and updates in non blocking manner.

5 Conclusions

The purpose of this project was to show event driven programming. The events are the transmission and reception of the messages, because of which event specific processes get triggered. The event driven runs the event loop called receive loop which run continuously until a message is received, when received it sends it to the event handler which is the incoming message processing function, and based on the topic it triggers further processes. The event driven programming is based on Publisher-Subscriber protocol, where the Sender publishes a message payload with the topic based on the Subscriber's topic, made for receiving by the Receiver.There are two processes which goes on the model, data generation and plotting streamed data. The publisher-subscriber protocol gets established with the handshaking mechanism where server and client notify each other by sending notification topics to each other. The client sends the payload topic 'connected' which informs the server that it is connected and then the server sends the client the payload topic 'me too' which notifies the client that the server is up and running and ready for accepting requests. Just as the client receives it, it initiates the data 'request' events and data streaming. When Client's receive loop gets into state of idle, it start its idle addition loop which publishes payload topic "request" again. The Server receives these messages, decodes it send it to the event handler. It triggers the if condition in the data generation loop and which generates and writes the data into the buffer.

A payload is then published with a subscriber topic called 'plotting'. The client receive loop which works every 0.5s receives the collected data, decodes it and sends it to event handler. It further is processed to be sent to the interactive plotter which has a Circular buffer. The event of receiving the message payload, triggers the actions sequentially. First the data gets stored in the Buffer, then it is read and sent for plotting. The plotting is carried by the flush events() and canvas draw() which keeps the interactive plot updated by processing the pending events (the data samples) that were queued up. In the end, when the if condition of ending the plot is met, the client sends a message with topic 'interrupt' which breaks the if condition in the data generation loop, thereby stopping any further sample generation. The model streams data without any loss or overwriting. Python banyan framework based on object oriented programming provides flexibility to build a messaging protocol based on our needs. The model shows the publisher-subscriber protocol is initialised with the handshaking mechanism. The timing of the data generation and plotting refresh time are well synchronised for optimum data transmission and reception. The experiment successfully showed the collection of data for 0.5 seconds and streaming of data to the plotting client, with client asking for request of data every time it is done with topic 'plotting'.