A LoRaWAN Simulator

Abstract

We have developed a discrete event, flexible, and to-be-open-sourced LoRa network simulator. Salient features of this simulator include the ability to control the simulator via an external script. The script of the developed simulator is written in Python. The developed simulator works at the IQ samples level at the PHY layer. MAC-PHY interaction is modeled. Node mobility is added in a modular approach. Several physical-layer demodulation mechanisms can be incorporated in a modular manner. Wireless propagation characteristics can be incorporated in a modular manner as well. This is a work in progress and shared here as a supporting document for our paper revision. This simulator does not currently have the downlink transmissions. We will be making this simulator open-source and allow the research community to contribute and add more features.

1 Introduction

We have developed a light-weight, flexible, and open-source simulator for LoRaWAN functionality at IQ sample level. The simulator is developed in Python. We consider the LoRaWAN architecture network which consists of many mobile nodes and gateways. The simulator uses modular approach to incorporate different models for the following:

1. Mobile node location and mobility
2. Mobile nodes' data generation process
3. Mobile nodes' channel access mechanism
4. Wireless channel
5. Signal processing at the receiving gateways

Further, we have incorporated the following features

1. Controllable via external script
2. Scalable
3. Works at Physical layer samples
4. Operates at MAC+PHY layer
5. Allows for ephemeral sessions (nodes arriving and departing)
6. Configurable SF and other LoRa parameters associated with a transmission

Draft (WIP)

1.1 Simulation Process

The working of the developed simulator is shown in the fig.1. There are mainly two parts of the developed simulator 1) External script 2) Event script as shown in fig.1. The External script controls the Event script through the number of events in the Event script. Basically, External script gives a configurable amount of time for the simulation of Event script. In the allotted time, Event script does the simulation for its generated event and at the end of the allotted time, Event script saves all the required context which would be useful when Event script will get another allotted slot for its simulation through the External script. In this way, the simulator uses the result of the previous Event script's simulation into the current simulation of the Event script. Hence, the flow of the simulation of the events will be maintained.

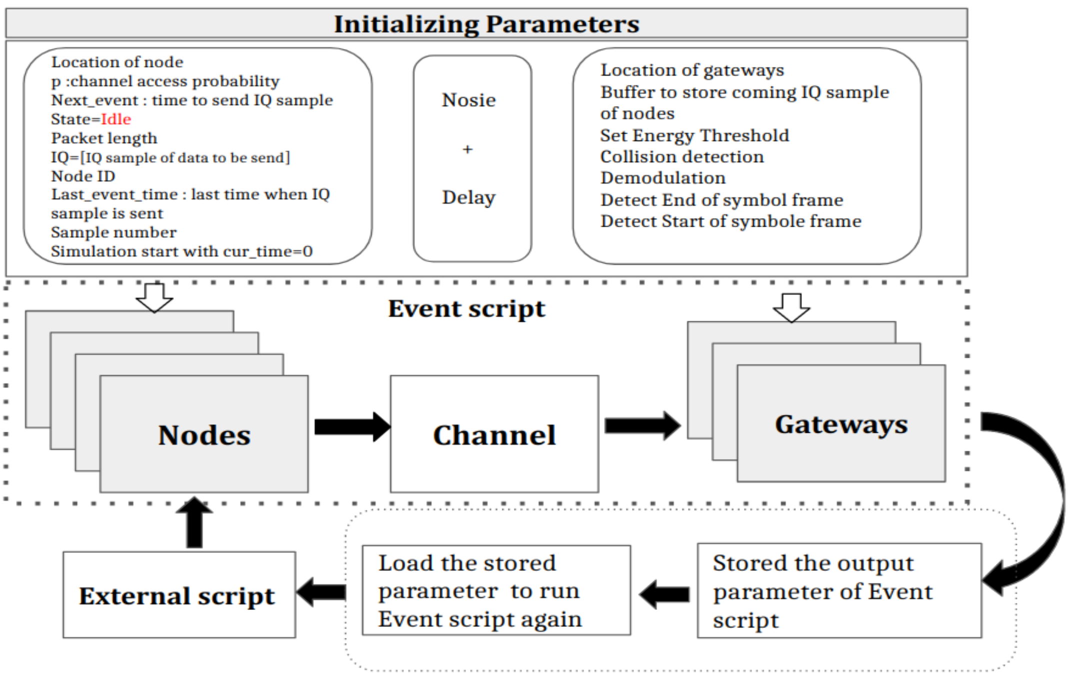

Figure 1: Overview of the developed simulator for LoRaWAN.

Figure 2 provides an overview of a receiver (gateway) getting IQ samples and the MAC data transmission attempt modeling. The figure considers two gateways and two mobile devices and shows how the mobile devices' transmissions are translated into the received IQ samples at the gateway and highlights the various pluggable (and configurable) modules involved in this process.

Figure 3 provides another control flow view of the simulator.

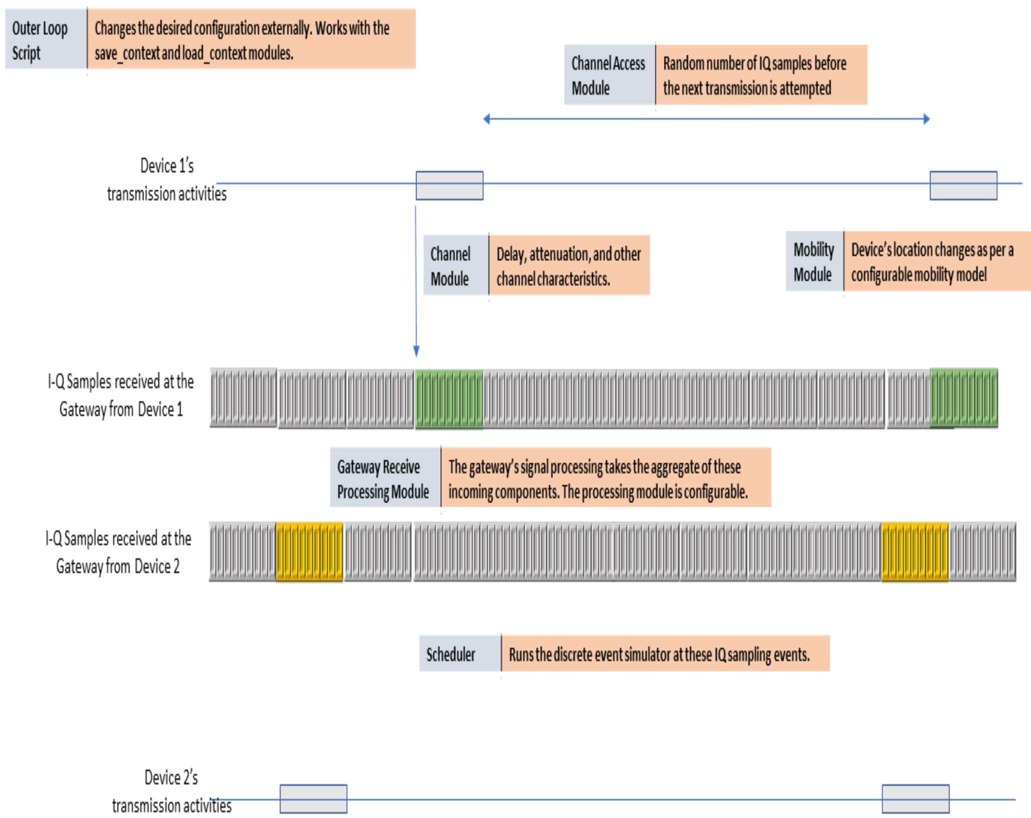

Figure 2: IQ sample processing of the developed simulator for LoRaWAN at PHY layer.

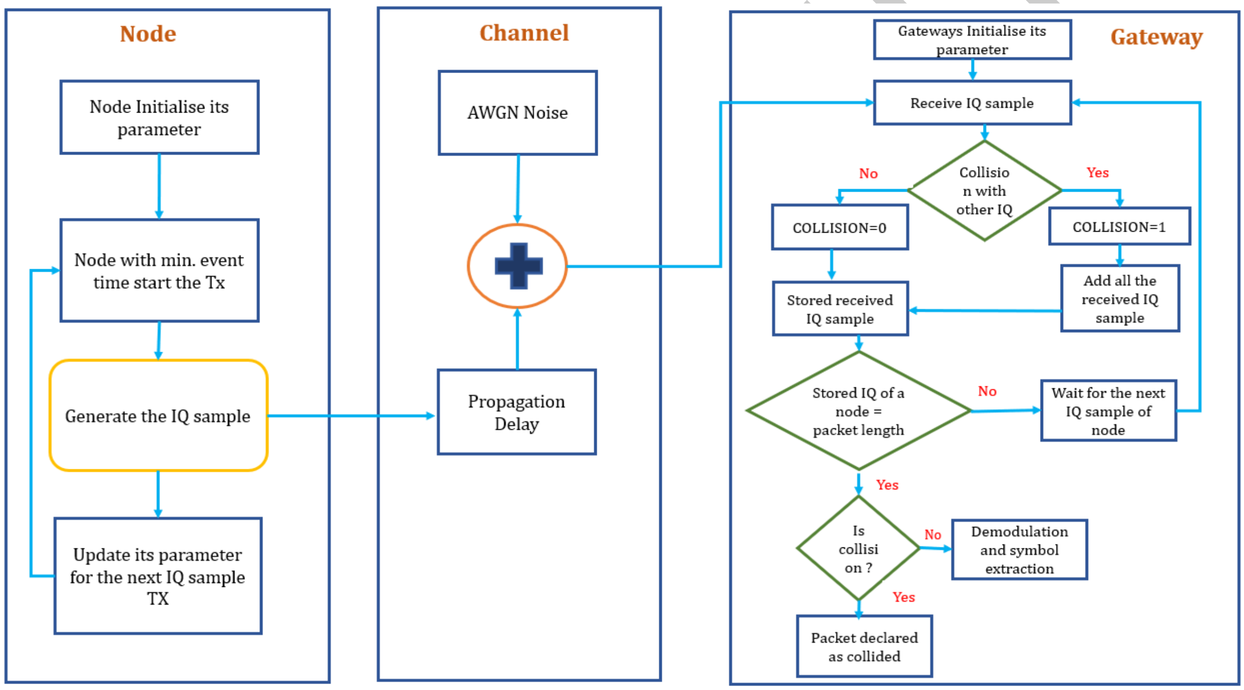

Figure 3: Flow diagram of the developed simulator.

```python
#WLOG, the sample duration is normalized to 1
#the access probabilities will be determined based on the slots
#the channel "attempt rate" will be determined by the sample interval as well

from bitstring import BitArray, BitStream
import math
import numpy as np
from scipy.stats import geom
from scipy.fftpack import fft
import os

NumGW = 1;
NumDev=20;
PacketLenBytes=40;

theta_m=30;

target_thinning_prob=.8

a_i=0
a_up=[]
number_of_preamble=7
preamble_up=[]
I_Q_sample_physical_layer_preamble=[]

log_enabled = {}
log_enabled["NODE"]=0
log_enabled["GW"]=0
log_enabled["MAIN"]=1

def print_log(device,*argv):
    if log_enabled[device]==1:
        print(device,end ="")
        print(":\t",end=" ")
        for arg in argv:
            print(arg,end =" ")
        print()

def save_context(varname,varvalue):
    filename="SavedVars/"+varname
    f=open(filename, "w")
    f.write(varvalue)
    f.close()

def load_context(varname,defaultvalue):
    filename="SavedVars/"+varname
    if os.path.exists(filename):
        f=open(filename, "r")
        return(f.read())
        f.close()
    else:
        return(defaultvalue)

def MAC_PHYSICAL_LAYER_PACKET(mac_payload_size,SF,mac_payload_stream=None):
    if mac_payload_stream==None:
        mac_payload_stream = BitArray(mac_payload_size) ##ba change## #generate bitstream of length mac_payload_size
    #chopping the mac bit-stream into packets of SF length for LoRa modulation
    step=0
    array_physical_symbol_bit=[]
    array_physical_symbol_decimal=[]
    I_Q_sample_physical_layer=[]
    M=2**SF
    for i in range(int(mac_payload_size/SF)):
        array_physical_symbol_bit.append(mac_payload_stream[step:step+int(SF)])
        step=int(SF)+step

```

```
    #converting the each pysical layer packet bit-stream into its decimal equivalent
    for j in range(len(array_physical_symbol_bit)):
        i=0
        for bit in array_physical_symbol_bit[j]:
            i=(i<<1) |bit
        array_physical_symbol_decimal.append(i)

    #print ("LoRa_symbol at physical layer without preamble",array_physical_symbol_decimal)

    # modulating each physical packet symbol with up-chrips
    a_up=array_physical_symbol_decimal
    #preamble aadition in mac payload at physical layer in order to send in air
    for i in range(number_of_preamble):
        for n in range(int(M)):
            preamble_up.append(np.exp(1j*2*np.pi*(((n**2)/(2*M))+((a_i/M)-.5)*n)))

    for i in range(len(a_up)): #for each symbol
        Lora_signal_up1=[]
        for n in range(int(M)):

            Lora_signal_up1.append(np.exp(1j*2*np.pi*(((n**2)/(2*M))+((a_up[i]/M)-.5)*(n))))

            I_Q_sample_physical_layer.append(np.exp(1j*2*np.pi*(((n**2)/(2*M))+((a_up[i]/M)-.5)*(n)))) #collecting total I/Q samples of physical layer packet

    I_Q_sample_physical_layer_preamble.append(preamble_up+I_Q_sample_physical_layer)

    return I_Q_sample_physical_layer

def LoRa_Receiver_demodulation(I_Q_sample_physical_layer,SF):
    Received_packet_IQ=[]
    #dechriping_lora_up1=[]
    Lora_up_conjugate1=[]
    step1=0
    a_i=0
    M=2**SF
    received_symbol=[]
    received_symbol_bits=[]
    received_symbol_bits1=[]
    received_symbol_bits2=[]
    mac_payload_at_receiver=[]

    for i in range(int(len(I_Q_sample_physical_layer)/(M))):
        Received_packet_IQ.append(I_Q_sample_physical_layer[step1:step1+int(M)])
        step1=step1+int(M)
    #print("eee",len(Received_packet_IQ))
    for i in range(len(Received_packet_IQ)):
        dechriping_lora_up1=[]
        for n in range(int(M)):

            Lora_up_conjugate1.append(np.exp(-1j*2*np.pi*(((n**2)/(2*M))+((a_i/M)-.5)*n)))
            dechriping_lora_up1.append(Received_packet_IQ[i][n]*Lora_up_conjugate1[n])
        d_fft=fft(dechriping_lora_up1)
        maximum_fre=np.argmax(d_fft)
        received_symbol.append(maximum_fre)
    #print("Received symbol at LoRa receiver",received_symbol)
    for i in range(len(received_symbol)):
        received_symbol_bits.append(bin(received_symbol[i]))
        received_symbol_bits1.append(received_symbol_bits[i][2:])
        received_symbol_bits2.append(received_symbol_bits1[i].zfill(int(SF)))
    mac_payload_at_receiver.append("".join(received_symbol_bits2))
    #print("mac_payload_at_receiver",mac_payload_at_receiver)
    #return mac_payload_at_receiver[0]
    return received_symbol
```

```
class Node():
    #initializes the location and other parameters of the node
    def __init__(self,space="ring",mobility="SRW",num=1):#for symmetric random walk
        strn="node"+str(num)+"loc"
        self.loc=int(load_context(strn,int(np.ceil(np.random.randint(0,359)/theta_m)))); #initial angle, in theta_m units
        #print_log("NODE","Initial Location ",self.loc);
        self.mobilityscale=15000000; #mobilityscale is in terms of samples. For each mobilityscale number of samples, the node moves left or right with equal probability
        #this is also the scale at which next transmission probabilities are decided
        strn="node"+str(num)+"p"
        #self.p=float(load_context(strn,np.random.uniform(target_thinning_prob/2.0,target_thinning_prob))); #probability of transmitting in a sample duration
        self.p=float(load_context(strn,np.random.uniform(0.1,0.8))); #probability of transmitting in a sample duration
        #print(self.p)
        #the above initialization should be less than the target thinning probability as target thinning probability is upper bounded by p
        strn="node"+str(num)+"next_event"
        self.next_event=int(load_context(strn,self.mobilityscale*(np.random.uniform(1,1.05))*geom.rvs(self.p))); #gets the first value for tranmission slot. staggers the exact transnmission slot to avoid inter-node synchronization
        #this is not the global time. this is time-to-next-event
        self.state="IDLE"; #better handling with FSM is required here
        self.samplenum=0;  #the ongoing IQ sample number
        self.num=num;
        strn="node"+str(num)+"num_attempts"
        self.num_attempts=int(load_context(strn,1));
        #2 is added to the length to ensure that the begining and end
        #are zero so that the receiver can perform energy detection.
        payload= BitArray(int=self.get_loc(),length=16)
        for temp in range(PacketLenBytes):
            payload=payload+BitArray(int=0,length=16)
        payload=payload+BitArray(int=self.num,length=16)

        #print("payload ",payload)

        y=MAC_PHYSICAL_LAYER_PACKET(mac_payload_size=len(payload),SF=8,mac_payload_stream=payload)
        #print(len(y))
        self.pktlen=len(y)+2; #assume len(y) IQ samples per physical layer transmission.
        self.IQ=(0+0j)*np.ones(self.pktlen); #replace this by IQ samples
        #print("length... ",len(self.IQ))
        self.IQ[1:len(y)+1]=y;
        strn="node"+str(num)+"last_event_time"
        self.last_event_time=int(load_context(strn,0));
        #print_log("NODE","Initial next event schedule",self.last_event_time+self.next_event);

        #print(self.next_event)

    def get_node_num(self):
        return self.num

    def get_next_time(self):
        return self.next_event

    def do_event(self):
        self.change_loc(self.next_event); #self.next_event is the last time interval
        self.last_event_time=self.last_event_time+self.next_event;#current time
        #print("last event time of node*********",self.last_event_time)
        if self.state=="IDLE": #next step is transmission
```

```
            self.state="Tx";
            self.samplenum=1;
            print_log("NODE", "attempt no. ",self.num,self.num_attempts,self.loc,self.last_event_time)
            self.next_event=1; #next event is IQ sample transmission again
        else:
            if self.state=="Tx":
                if self.samplenum==self.pktlen: #last packet
                    #print("%%%%%%%% samplenum",self.samplenum)
                    self.state="IDLE"; #better handling with FSM is required here
                    self.next_event=int(self.mobilityscale*(np.random.uniform(1,1.05))*geom.rvs(self.p)); #gets the first value for tranmission slot. staggers the exact transnmission slot to avoid inter-node synchronization
                    #self.next_event=self.mobilityscale*geom.rvs(self.p);#at the scale of mobilityscale (number of samples)
                    self.cur_loc=self.get_loc()
                    print_log("NODE", "Going to Idle...",self.num,self.last_event_time,self.cur_loc);
                    self.change_loc(self.next_event)
                    payload= BitArray(int=self.get_loc(),length=16)
                    for temp in range(PacketLenBytes):
                        payload=payload+BitArray(int=0,length=16)
                    payload=payload+BitArray(int=self.num,length=16)
                    y=MAC_PHYSICAL_LAYER_PACKET(mac_payload_size=len(payload),SF=8,mac_payload_stream=payload)
                    self.IQ[1:len(y)+1]=y;
                    self.samplenum=0;
                    #print("before next attempt",self.num_attempts,self.num)
                    self.num_attempts=self.num_attempts+1;

                else: #not transiting to IDLE
                    self.state="Tx";
                    self.samplenum=self.samplenum+1;
                    self.next_event=1; #next event is IQ sample transmission again

    def get_state(self):
        return self.state;

    def get_samplenum(self):
        return self.samplenum;

    def get_iq(self,num):
        if num<self.pktlen:
            return self.IQ[num];
        else:
            return 0+0j; #nothing to be sent when going to idle. this should never happen

    def get_pktlen(self):
        return(self.pktlen);

    def get_loc(self):
        return(self.loc);

    def change_loc(self,time):
        for i in range(time//self.mobilityscale):#get time/mobilityscale number of transitions
            if np.random.random()<0.5:
                self.loc=(self.loc + 1)%int(360/theta_m)
            else:
                self.loc=(self.loc - 1)%int(360/theta_m)

    def get_last_event_time(self):
        return(self.last_event_time)
```

```
class GW():
    #initializes the location and other parameters of the node
    def __init__(self):#for symmetric random walk
        strn="gateway_loc"
        self.loc=int(load_context(strn,np.random.randint(0,359))); #Fixed Locations
        self.iq=[];
        self.rx=[];
        self.energy_threshold=0.5;# the energy threshold for detection
        self.frame_ongoing=0; #to differentiate start of frame from end of frame
        self.current_iq_train=[];
        self.is_collision=0;
        self.decoded=0;
        self.was_collision=0;
        self.node_sample_count=1000000;
        self.node_num=55
        self.num_sample_current_instant=0

    def start_receiving_iq(self): #means a new event has happened
        self.num_sample_current_instant=0;#reset for the next sample
        self.iq.append(0+0j);
        #print_log("GW", "initialized iq",self.iq);

    def receive_iq(self,loc,source_iq,node): #add iq component to the currently received sample
        #loc is the location of the sender node. this is to get the channel
        if abs(source_iq)>self.energy_threshold:
            self.num_sample_current_instant=self.num_sample_current_instant+1;#count the number of transmitters
        if self.num_sample_current_instant>1:
            if self.is_collision==0:
                self.is_collision=1;
                print_log("GW",".....................collision detected in GW");
        self.iq[-1]=self.iq[-1]+self.channel(loc)*source_iq

    def noise(self):
        return(0+0j); #AWGN to be added

    def channel(self, loc):
        return(1); #to add path loss in this later

    def stop_receiving_iq(self):
        #add AWGN to the received signal
        #discard the iq samples received in this instant if energy is not detected
        #all the decoding state machine goes here
        #print("received iq sample",self.iq[-1]);
        #this will require use to have zero added at the begining of transmission and reception so that
        #the receiver can do energy detection. Else, there will always be energy on the channel

        if self.num_sample_current_instant>0:
            self.current_iq_train.append(self.iq[-1])
            if self.frame_ongoing==0:
                print_log("GW", "start of a new frame");
                self.frame_ongoing=1;
        else: #means an idle sample
            print_log("GW", "an Idle sample found");
            if self.frame_ongoing==1:
                print_log("GW", "Tx to Idle transition");
                self.frame_ongoing=0; #get ready for detecting the next start of frame
                self.was_collision=self.is_collision;
                if self.is_collision==0:

                    self.rx=LoRa_Receiver_demodulation(I_Q_sample_physical_layer=self.current_iq_train,SF=8)
                    self.decoded=1;
                else:
```

```
                    self.rx=[]; #to ensure that the previous decoded value is not carried over
                self.is_collision=0; #reset so that next frame starts with no collision assumption
                del(self.current_iq_train);
                self.current_iq_train=[];
        del(self.iq)
        self.iq=[];

        if self.was_collision==1: #means an idle sample and also a collision
            self.was_collision=0;
            return("collided");
        if self.decoded==1: #print the message that was received and decoded, when no collision
            self.decoded=0;
            print_log("GW", "decoded: ",self.rx)
            return(self.rx);

def find_entropy(sequence):
    elements=[];
    for i in sequence:
        if i not in elements:
            elements.append(i);
    tpm=[[0 for i in range(len(elements))] for j in range(len(elements))];
    number=[0 for i in range(len(elements))];
    for i in range(len(sequence)-2):
        tpm[sequence[i]][sequence[i+1]]=tpm[sequence[i]][sequence[i+1]]+1;
        number[sequence[i]]=number[sequence[i]]+1;
    for i in elements:
        for j in elements:
            tpm[i][j]=tpm[i][j]/number[i]

    entropy=0.0;
    for i in elements:
        for j in elements:
            entropy=entropy+tpm[i][j]*math.log(tpm[i][j]);


def find_thinning_prob(sucess,attempt):
    return(sucess/attempt)


#generate the nodes
nodes=[Node(num=i) for i in range(NumDev)]

gws=[GW() for i in range(NumGW)]
p_history=[[.3],[.3],[.3],[.3]]
# following does the scheduling part

cur_time=int(load_context("cur_time",0));

loc_est=[[] for i in nodes]

num_received=[0 for i in nodes];

for j in nodes:
    strn="node"+str(j.num)+"num_received"
    num_received[j.num] = int(load_context(strn,0));


y=0

max_num_events=3000000

if cur_time==0:
    for i in nodes:
        print_log("MAIN",",", cur_time,",",i.p,",","not
```

```
        known",",",i.num_attempts,",",i.num)

for i in range(max_num_events): #number of events
    # if y=="collided":
    #     print("&&&&")
    #print("**************",i)
    time_to_next_event=10000000;

    for j in gws:
        j.start_receiving_iq();#new event has happened, add an IQ element to the array at the receiver

    for j in nodes:
        if j.get_last_event_time()+j.get_next_time()<cur_time+time_to_next_event:

            #print("j.get_last_event_time().....",j.get_last_event_time(),j.get_next_time(),j.num)
            time_to_next_event=j.get_last_event_time()+j.get_next_time()-cur_time;#error
    cur_time=cur_time+time_to_next_event;
    iq=0;

    for j in nodes:
        if j.get_last_event_time()+j.get_next_time()==cur_time:
            for g in gws: #this can be modified to include the neighbor set
                g.receive_iq(source_iq=j.get_iq(j.get_samplenum()),loc=j.get_loc(),node=j.get_node_num());#new event has happened, add an IQ element to the array at the receiver
            j.do_event();

    for j in gws:
        y=j.stop_receiving_iq();#new event has happened, add an IQ element to the array at the receiver
        if y!=None:#means this was the last IQ sample
            #if y!="collided" and nodes[y[-1]].samplenum==1026:
            if y!="collided":
                sending_node=y[-1];
                #print_log("MAIN", "One event ended with success",cur_time,sending_node)
                num_received[sending_node]=num_received[sending_node]+1;
                #get the decoded message, estimate entropy etc and update the probability for this node
                #the id of the node is known as part of the message received
                if num_received[sending_node]%10 == 0:

                    thinning_probability=(nodes[sending_node].p)*find_thinning_prob(num_received[sending_node],nodes[sending_node].num_attempts);

                    if abs(target_thinning_prob-thinning_probability)>0.05:
                        #print_log("MAIN",",",cur_time,",",nodes[sending_node].p,",",thinning_probability,",",nodes[sending_node].num_attempts,",",y[-1]);
                        #print_log("NODE","target achieved")
                    #else:
                        if thinning_probability<target_thinning_prob:
                            #print_log("MAIN","target is more")

                            nodes[sending_node].p=nodes[sending_node].p+0.1*(target_thinning_prob-thinning_probability) #Increase
                        else:
                            #print_log("MAIN","target is less")

                            nodes[sending_node].p=nodes[sending_node].p+0.1*(target_thinning_prob-thinning_probability) #decrease
                        if nodes[sending_node].p<0.00005:
                            nodes[sending_node].p=0.00005
                        if nodes[sending_node].p>0.9:
                            nodes[sending_node].p=0.9
                    print_log("MAIN",",",
```

```
                            cur_time,",",nodes[sending_node].p,",",thinning_probability,",",nodes
                            [sending_node].num_attempts,",",y[-1])
                            nodes[sending_node].num_attempts=0
                            num_received[sending_node]=0
                    #else:
                        #print_log("MAIN",
                        "**********************COLLISION*******************************");

        #exit should be at the end when the event before this IDLE event is processed
        if i>int(0.5*max_num_events):
            idle=1
            for j in nodes:
                if j.state!="IDLE":
                    idle=0;
            if idle==1:
                #print_log("MAIN","System found to be idle  ", cur_time);
                break;

    save_context("cur_time",str(cur_time));
    #print("cur_time",cur_time)
    for j in nodes:
        strn="node"+str(j.num)+"loc"
        #print("loc... ",j.get_loc())
        save_context(strn,str(j.get_loc()));
        strn="node"+str(j.num)+"p"
        save_context(strn,str(j.p));
        strn="node"+str(j.num)+"next_event"
        save_context(strn,str(j.next_event));
        strn="node"+str(j.num)+"state"
        save_context(strn,str(j.state));
        strn="node"+str(j.num)+"last_event_time"
        #print("lat even at node
        side*********************************+++++",j.last_event_time)
        save_context(strn,str(j.last_event_time));
        strn="node"+str(j.num)+"num_attempts"
        save_context(strn,str(j.num_attempts));
        strn="node"+str(j.num)+"num_received"
        save_context(strn,str(num_received[j.num]));

    for g in gws:
        strn="gateway_loc"
        #print("gateway location",g.loc)
        save_context(strn, str(g.loc))
```