main.cpp

```
/**************Dynamic Buffer Allocation**********************/

/*****************Syetem Header Files********************/
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <string>

/*****************My Header Files***********************/
#include "macro_def.h"
#include "my_func.h"
#include "my_struc.h"


/*****************global variables********************/

long seed[1];

struct Cell **cell;

int buffer_bound;

//????
int source_queue_max_length;
int sum_relay_queue_max_length;

int ?;

float alpha;

int n;
int m;


int tagged_S=5;
int tagged_D=6;


/******************test quantities*******************/


int SD_num; //For testing the tagged flow direct transmission opportunity
int SR_num; //For testing the tagged flow S->R transmission opportunity
int RD_num; //For testing the tagged flow R->D transmission opportunity

int S_out_number; //For testing the output_opportunity of the local queue
int R_in_number; //For testing the input rate of the relay queue
int R_out_number; //For testing the output rate of the relay queue
long lost_number;

long tagged_S_source_queue_empty;
long tagged_D_source_queue_empty;

long tagged_S_buffer_empty;
long tagged_D_buffer_empty;

long tagged_S_buffer_full;
long tagged_D_buffer_full;

float delay;
float queuing_delay;
float delivery_delay;
```

```
/******************data collection******************/

ofstream ftest;


/*****************Notice! Project Entrance************/
int main()
{
    *seed=-time(0);

    /***************data collection******************/
    ftest.open("n_200_3.dat");


    /***************network settings*****************/
    n=200;
    m=10;

    buffer_bound=10;

    source_queue_max_length = 5;
    sum_relay_queue_max_length = 5;


    alpha=0.5;

    //????
    ?=5;

    //float mus=0.153;


    cout<<"n: "<<n<<" m: "<<m<<endl;

    /******************simulation settings***********/
    long time_max[15]= {0,20000000,20000000,20000000,20000000,20000000,20000000,20000000,20000000,20000000,
20000000,20000000,20000000,20000000,20000000}; //simulation runs for time_max slots

    long time_slot=-1; //slot_0, slot_1, slot_2, ......
    int round=1; //for each network setting, how many rounds a simulation has done
    float lambda[15]= {0,0.002,0.005,0.01,0.015,0.02,0.05,0.1,0.2,0.5,0.6,0.7,0.8,0.9,1};


    /***************building and initializing data structure******************/

    cout<< "System is allocating RAM resource for simulation!!!"<<endl<<endl;

    struct Node *node; //building nodes in the network
    node=new struct Node[n+1];//node[1],...,node[n]

    cell=new struct Cell *[m]; //
    for(int i=0; i<m; i++)
    {
        cell[i]= new struct Cell[m];
        for(int j=0; j<m; j++)
        {
            cell[i][j].node_in_cell=new int[n+1];
            cell[i][j].row=i;
            cell[i][j].col=j;
        }
    }

    /***************relay queue for other n-2 flows, the map between node[i].relay_queue[j] and its destined
    node_id is like this: if, node_id<the current node id index i, then, the current queue index j indicates
    the destined node_id; else if, node_id>index i, then, j=node_id-2*********************************/
```

```
    for(int i=1; i<=n; i++)
        node[i].relay_queue= new queue<struct Packet>[n-1];  //relay_queue[1], relay_queue[2],...,
relay_queue[n-2]

    string mobility_model;
    mobility_model="IID";

    cout<<"************************"<<mobility_model<<"*********************************" <<endl;
    cout<<" n="<<n<<" m="<<m<<" buffer-size="<<buffer_bound<<" transmission-ratio="<<alpha<<" No Feedback"<<endl
;
    cout<<" probNum="<<probNum<<" source_queue_max_length="<<source_queue_max_length<<"
sum_relay_queue_max_length="<<sum_relay_queue_max_length<<endl;
    cout<<"************************************************************" <<endl<<endl;

    ftest<<"************************"<<mobility_model<<"*********************************" <<endl;
    ftest<<" n="<<n<<" m="<<m<<" buffer-size="<<buffer_bound<<" transmission-ratio="<<alpha<<" No Feedback"<<
endl;
    ftest<<" probNum="<<probNum<<" source_queue_max_length="<<source_queue_max_length<<"
sum_relay_queue_max_length="<<sum_relay_queue_max_length<<endl;
    ftest<<"************************************************************" <<endl<<endl;


    /***********we do simulation as input rate lambda approaches capacity mu********************/
    /********************we just focus on a tagged node pair************************************/

    for(round=1; round<=14; round++)
    {
        //simulation initialization

        cout<<"***********new round:"<<round<<" lambda: "<<lambda[round]<<"************"<<endl;
        //cout<<"***********new round:"<<round<<" lamda: "<<lambda<<endl;

        //ftest<<"***********new round:"<<round<<" lamda: "<<lambda<<endl;

        ftest<<"***********new round:"<<round<<" lambda: "<<lambda[round]<<"************"<<endl;


        cout<<" System is initializing simulation status for round: "<<round<<endl;

        //initialize test statistic variables


        SD_num=0;
        SR_num=0;
        RD_num=0;


        S_out_number=0;
        R_in_number=0;
        R_out_number=0;
        lost_number=0;

        queuing_delay=0;
        delivery_delay=0;
        delay=0;

        tagged_S_source_queue_empty=0;
        tagged_D_source_queue_empty=0;
        tagged_S_buffer_empty=0;
        tagged_D_buffer_empty=0;
        tagged_S_buffer_full=0;
        tagged_D_buffer_full=0;

        //initialize each node
        for(int i=1; i<=n; i++)
        {
```

```
    node[i].row=-1; //node position
    node[i].col=-1;

    //clear all queues
    while(!(node[i].local_queue.empty())) //clear local queue
        node[i].local_queue.pop();

    for(int j=0; j<n-1; j++) //clear relay queue
    {
        while(!(node[i].relay_queue[j].empty()))
            node[i].relay_queue[j].pop();
    }
    node[i].source_queue_length=0;
    node[i].sum_relay_queue_length=0;

    node[i].arrival_ct=0;
    node[i].recv_ct=0;
}


//initialize time clock
time_slot=-1;

cout<< " Simulation is starting !!! "<<endl;

//the main simulation body begins here
while(time_slot<time_max[round])
{
    time_slot++;

    update_node_position_IID(n, m, node);
    //update_node_position_RWalk(n, m, node);
    //update_node_position_RWaypoint(n, m, node);

    collect_nodes_per_cell(n, m, node);

    THROR(node, time_slot); //???????

    //we locally generate packets for all n source nodes, only when time_slot==next_arrival_time
    for(int i=1; i<=n; i++)
    {

        if(probabilityP(lambda[round]))//a new packet arrives in this time slot for node i
        {
            node[i].arrival_ct++;

            if(node[i].source_queue_length < source_queue_max_length)
            {
                struct Packet packet;
                packet.id=node[i].arrival_ct;
                packet.arrival_time=time_slot;
                packet.reception_time=0;

                if(node[i].local_queue.empty())
                {
                    packet.queue_head_time=time_slot;
                }
                else
                {
                    packet.queue_head_time=0;
                }

                node[i].local_queue.push(packet);
                node[i].source_queue_length++;
            }
            else
```

```
                {
                    if(i==tagged_S)
                    {
                        lost_number++;
                    }
                }
            }
        }

        //H2HR(node, time_slot); //???????


        if (node[tagged_S].source_queue_length == 0)
        {
            tagged_S_source_queue_empty++;
        }

        if (node[tagged_D].source_queue_length == 0)
        {
            tagged_D_source_queue_empty++;
        }

        if (node[tagged_S].source_queue_length + node[tagged_S].sum_relay_queue_length == 0)
        {
            tagged_S_buffer_empty++;
        }

        if (node[tagged_D].source_queue_length + node[tagged_D].sum_relay_queue_length == 0)
        {
            tagged_D_buffer_empty++;
        }

        if (node[tagged_S].source_queue_length + node[tagged_S].sum_relay_queue_length == buffer_bound)
        {
            tagged_S_buffer_full++;
        }

        if (node[tagged_D].source_queue_length + node[tagged_D].sum_relay_queue_length == buffer_bound)
        {
            tagged_D_buffer_full++;
        }

    }


    cout<<" node "<<tagged_S<<" S-D transmission opportunity: "<<1.0*SD_num/time_slot<<endl;

    cout<<" node "<<tagged_S<<" S-R transmission opportunity: "<<1.0*SR_num/time_slot<<endl;

    cout<<" node "<<tagged_S<<" R-D transmission opportunity: "<<1.0*RD_num/time_slot<<endl;

    cout<<" node "<<tagged_S<<" generates "<<node[tagged_S].arrival_ct<<" packets in "<<time_slot<<" time
slots. Input rate: "<<1.0*node[tagged_S].arrival_ct/time_slot<<endl;

    cout<<" node "<<tagged_D<<" receives "<<node[tagged_D].recv_ct<<" packets in "<<time_slot<<" time slots.
Throughput rate: "<<1.0*node[tagged_D].recv_ct/time_slot<<endl;

    cout<<" node "<<tagged_D<<" receives/generates: "<<1.0*node[tagged_D].recv_ct/node[tagged_S].arrival_ct
<<endl;

    cout<<" node "<<tagged_S<<" loses "<<lost_number<<" packets in "<<time_slot<<" time slots. Packet lost
rate: "<<1.0*lost_number/time_slot<<endl;

    cout<<" node "<<tagged_S<<" the output opportunity of the local queue is: "<<S_out_number<<" "<<1.0*
S_out_number/time_slot<<endl;
```

```
        cout<<" node "<<tagged_S<<" the input rate of the relay queue is: "<<R_in_number<<" "<<1.0*R_in_number/
time_slot<<endl;

        cout<<" node "<<tagged_D<<" the output rate of the relay queue is: "<<R_out_number<<" "<<1.0*
R_out_number/time_slot<<endl;

        cout<<" node "<<tagged_S<<" source queue is empty with probability "<<1.0*tagged_S_source_queue_empty/
time_slot<<endl;

        cout<<" node "<<tagged_D<<" source queue is empty with probability "<<1.0*tagged_D_source_queue_empty/
time_slot<<endl;

        cout<<" node "<<tagged_S<<" buffer is empty with probability "<<1.0*tagged_S_buffer_empty/time_slot<<
endl;

        cout<<" node "<<tagged_D<<" buffer is empty with probability "<<1.0*tagged_D_buffer_empty/time_slot<<
endl;

        cout<<" node "<<tagged_S<<" buffer is full with probability "<<1.0*tagged_S_buffer_full/time_slot<<endl;

        cout<<" node "<<tagged_D<<" buffer is full with probability "<<1.0*tagged_D_buffer_full/time_slot<<endl;

        cout<<" node "<<tagged_S<<"'s average packet queuing delay: "<<queuing_delay<<endl;

        cout<<" node "<<tagged_S<<"'s average packet delivery delay: "<<delivery_delay<<endl;

        cout<<" node "<<tagged_S<<"'s average packet end-to-end delay: "<<delay<<endl<<endl<<endl;




        ftest<<" node "<<tagged_S<<" S-D transmission opportunity: "<<1.0*SD_num/time_slot<<endl;

        ftest<<" node "<<tagged_S<<" S-R transmission opportunity: "<<1.0*SR_num/time_slot<<endl;

        ftest<<" node "<<tagged_S<<" R-D transmission opportunity: "<<1.0*RD_num/time_slot<<endl;

        ftest<<" node "<<tagged_S<<" generates "<<node[tagged_S].arrival_ct<<" packets in "<<time_slot<<" time
slots. Input rate: "<<1.0*node[tagged_S].arrival_ct/time_slot<<endl;

        ftest<<" node "<<tagged_D<<" receives "<<node[tagged_D].recv_ct<<" packets in "<<time_slot<<" time
slots. Throughput rate: "<<1.0*node[tagged_D].recv_ct/time_slot<<endl;

        ftest<<" node "<<tagged_D<<" receives/generates: "<<1.0*node[tagged_D].recv_ct/node[tagged_S].arrival_ct
<<endl;

        ftest<<" node "<<tagged_S<<" loses "<<lost_number<<" packets in "<<time_slot<<" time slots. Packet lost
rate: "<<1.0*lost_number/time_slot<<endl;

        ftest<<" node "<<tagged_S<<" the output opportunity of the local queue is: "<<S_out_number<<" "<<1.0*
S_out_number/time_slot<<endl;

        ftest<<" node "<<tagged_S<<" the input rate of the relay queue is: "<<R_in_number<<" "<<1.0*R_in_number/
time_slot<<endl;

        ftest<<" node "<<tagged_D<<" the output rate of the relay queue is: "<<R_out_number<<" "<<1.0*
R_out_number/time_slot<<endl;

        ftest<<" node "<<tagged_S<<" source queue is empty with probability "<<1.0*tagged_S_source_queue_empty/
time_slot<<endl;

        ftest<<" node "<<tagged_D<<" source queue is empty with probability "<<1.0*tagged_D_source_queue_empty/
time_slot<<endl;
```

```
        ftest<<" node "<<tagged_S<<" buffer is empty with probability "<<1.0*tagged_S_buffer_empty/time_slot<<
endl;

        ftest<<" node "<<tagged_D<<" buffer is empty with probability "<<1.0*tagged_D_buffer_empty/time_slot<<
endl;

        ftest<<" node "<<tagged_S<<" buffer is full with probability "<<1.0*tagged_S_buffer_full/time_slot<<endl
;

        ftest<<" node "<<tagged_D<<" buffer is full with probability "<<1.0*tagged_D_buffer_full/time_slot<<endl
;

        ftest<<" node "<<tagged_S<<"'s average packet queuing delay: "<<queuing_delay<<endl;

        ftest<<" node "<<tagged_S<<"'s average packet delivery delay: "<<delivery_delay<<endl;

        ftest<<" node "<<tagged_S<<"'s average packet end-to-end delay: "<<delay<<endl<<endl<<endl;
    }

    cout<< " System is deleting RAM resources!!!"<<endl<<endl;


    for(int i=0; i<m; i++)
    {
        for(int j=0; j<m; j++)
            delete []cell[i][j].node_in_cell;
    }

    for(int i=0; i<m; i++)
        delete []cell[i];
    delete []cell;

    for(int i=1; i<=n; i++)
        delete []node[i].relay_queue;
    delete []node;


    ftest.close();

    cout << "Simulation is finished!" <<endl;
    int tmp=0;
    cin>>tmp;

    return 0;
}
```

my_struct.h

```
#ifndef MY_STRUC_H_INCLUDED
#define MY_STRUC_H_INCLUDED

#include <queue>

using namespace std;

/**************************data structure*************************/

struct Packet
{
    int id;              //packet indicator
    int arrival_time ;   //arrival time of this packet
    int queue_head_time;
    int reception_time ; // reception time of this packet
    //the time when this packet arrives its local queue
};
```

```
struct Node
{
    int row ; //the row id of this node in a m*m cell-partition network, the cell number C=m*m
    int col ; //the column id of this node

    queue <struct Packet >local_queue ;   //store locally generated packets
    queue <struct Packet > *relay_queue ; //n-2 relay queues to store packets for other traffic flows
    int source_queue_length; //
    int sum_relay_queue_length;

    //int source_queue_max_length; //??????
    //int sum_relay_queue_max_length;

    int arrival_ct ; //total self-generated packets at this node
    int recv_ct ;    //total received packets at this node as a destination
};



struct Cell
{
    int row; //the row id of this cell
    int col; //the column id of this cell

    int *node_in_cell; //recording the node id in this cell;
    int nodenum_of_cell; //recording the node number of this cell;
};

#endif // MY_STRUC_H_INCLUDED
```

my_func.h

```
#ifndef MY_FUNC_H_INCLUDED
#define MY_FUNC_H_INCLUDED

float ran0_1(long  *idum);

int probabilityP(float p);

void update_node_position_IID(int n,int m, struct Node *node);

void update_node_position_RWalk(int n,int m, struct Node *node);

void update_node_position_RWaypoint(int n,int m, struct Node *node);

void collect_nodes_per_cell(int n, int m, struct Node *node);

void SDtrans(struct Node *node, long time_slot, int trans_id, int dest_id);

void SRtrans(struct Node *node, long time_slot,int trans_id, int recv_id);

void RDtrans(struct Node *node, long time_slot,int trans_id, int recv_id);

void THROR(struct Node *node, long time_slot);

#endif // MY_FUNC_H_INCLUDED
```

macro_def.h

```
#ifndef MACRO_DEF_H_INCLUDED
#define MACRO_DEF_H_INCLUDED

#define IM1 2147483563
#define IM2 2147483399
```

```
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
#define PI 3.141592654



#endif // MACRO_DEF_H_INCLUDED
```

ran0_1.cpp

```
#include "macro_def.h"

float ran0_1(long  *idum )
{
    int j;
    long k;
    static  long idum2 =123456789 ;
    static  long iy=0;
    static  long iv[NTAB ];
    float temp ;
    if (*idum <= 0)
    {
        //Initialise.
        if (-(* idum ) < 1) *idum =1;  //Be sure to prevent 'idum' = 0.
        else  *idum = -(* idum );
        idum2 =(* idum );
        for  (j=NTAB +7; j>=0; j--)
        {
            //Load the shuffle table (after 8 warmups).
            k=(* idum )/IQ1;
            *idum =IA1*(*idum -k*IQ1)-k*IR1;
            if (*idum < 0) *idum +=IM1;
            if (j <NTAB )iv[j] = *idum ;
        }
        iy =iv[0];
    }
    k= (*idum )/IQ1; //Start here when not initialising.
    *idum =IA1*(*idum -k*IQ1)-k*IR1; //Compute 'idum=(IA1*idum)' % IM1
    if (*idum < 0) *idum +=IM1;//without overflows by Schrage's method.
    k=idum2 /IQ2;
    idum2 =IA2*(idum2 -k*IQ2)-k*IR2; //Compute 'idum2=(IA2*idum)' % IM2
    if (idum2 < 0)idum2 +=IM2;
    j=iy/NDIV; //Will be in the range 0_NTAB-1.
    iy=iv[j]-idum2 ; //Here 'idum' is shuffles, 'idum' and
    //'idum2' are combined to generate output.
    iv[j] = *idum ;
    if (iy < 1)iy +=IMM1;
    if ((temp =AM*iy) >RNMX) return RNMX; //Because users don't expect endpoint values.
    else  return temp ;
}
```

update_node_position.cpp

```
#include "my_func.h"
```

```c
#include "my_struc.h"

extern long seed[1];


/**
update the position of each node at the beginning of each time slot
according to the i.i.d mobility model
**/

void update_node_position_IID(int n,int m, struct Node *node)
{
    int row =0, col =0;

    for(int i=1; i<=n; i++)
    {
        row=(int)(m*ran0_1(seed)); // random select a row id among 0,1,...,m-1 with equal probability
        col=(int)(m*ran0_1(seed)); // random select a column id among 0,1,...,m-1 with equal probability
        node[i].row=row;
        node[i].col=col;
    }
}


/**
update the position of each node at the beginning of each time slot
according to the random walk mobility model
**/
void update_node_position_RWalk(int n,int m, struct Node *node)
{
    int horizontal_move=0; //-1,0,1
    int vertical_move=0;//-1,0,1

    for(int i=1; i<=n; i++)
    {
        horizontal_move=(int)(3*ran0_1(seed))-1;
        vertical_move=(int)(3*ran0_1(seed))-1;
        node[i].row=(node[i].row+vertical_move+m)%m;
        node[i].col=(node[i].col+horizontal_move+m)%m;
    }
}

/**
update the position of each node at the beginning of each time slot
according to the random way point mobility model
**/
void update_node_position_RWaypoint(int n,int m, struct Node *node)
{
    int v_x=0,v_y=0; //velocity
    int d_x=0,d_y=0; //direction

    for(int i=1; i<=n; i++)
    {
        //determine move direction
        d_x=(int)(2*ran0_1(seed));//0,1
        d_y=(int)(2*ran0_1(seed));//0,1
        if(d_x==0)d_x=-1;
        if(d_y==0)d_y=-1;

        //determine speed
        v_x=(int)(3*ran0_1(seed))+1;//1,2,3
        v_y=(int)(3*ran0_1(seed))+1;//1,2,3

        node[i].row=(node[i].row+d_y*v_y+m)%m;
        node[i].col=(node[i].col+d_x*v_x+m)%m;
    }
```

```
}
```

collect_nodes_per_cell.cpp

```
#include "my_struc.h"

extern struct Cell **cell;

/**
collect nodes in each cell
**/
void collect_nodes_per_cell(int n, int m, struct Node *node)
{
    //reset the state of each cell
    for(int i=0; i<m; i++)
    {
        for(int j=0; j<m; j++)
            cell[i][j].nodenum_of_cell=0;
    }

    int r_d=0; //recording the difference between the node row id and the cell row id
    int c_d=0; //recording the difference between the node column id and the cell column id

    int flag=0; // 0: the node is not in this cell 1: the node is the cell

    int index=0; // recording the node is in this cell

    for(int i=1; i<=n; i++)
    {
        flag=0;

        for(int s=0; s<m; s++)
        {
            for(int t=0; t<m; t++)
            {
                r_d=node[i].row-cell[s][t].row;
                c_d=node[i].col-cell[s][t].col;

                if((r_d==0)&&(c_d==0)) //node i is in an active cell
                {
                    index=cell[s][t].nodenum_of_cell;
                    cell[s][t].node_in_cell[index]=i;
                    cell[s][t].nodenum_of_cell++;

                    flag=1;
                    break;
                }
            }

            if(flag==1)
                break;
        }
    }
}
```

probabilityP.cpp

```
#include "my_func.h"
#include <iostream>

using namespace std;


extern long seed[1];
```

```
int probabilityP(float p)
{
    float sim=ran0_1(seed);
    if(sim<p)
        return 1;
    else
        return 0;
}
```

THROR.cpp

```
#include "my_struc.h"
#include "my_func.h"

extern long seed[1];

extern struct Cell **cell;

extern int tagged_S;
extern int tagged_D;

extern int n;
extern int m;


extern int SD_num;
extern int SR_num;
extern int RD_num;
extern int S_out_number;

extern float alpha;

extern int ?;

extern int sum_relay_queue_max_length;

/*******************************************************************************************
                                Traffic Setting
                1-->2, 2-->3,...,i-1-->i,...,n-1-->n, n-->1
         with out loss of generality,we focus on a tagged node pair
*******************************************************************************************/

void THROR(struct Node *node, long time_slot)
{


    int dest_id=0; //indicate the direct destination node id
    int trans_id=0; //randomly selected transmitter
    int recv_id=0; //randomly selected receiver
    int index=0;

    int flag=0;

    int nodenum_of_cell=0;



    for(int i=0; i<m; i++)
    {
        for(int j=0; j<m; j++)
        {

            flag=0;

            nodenum_of_cell=cell[i][j].nodenum_of_cell;
```

```
            if(nodenum_of_cell>=2)
            {
                index=(int)(nodenum_of_cell*ran0_1(seed));

                trans_id=cell[i][j].node_in_cell[index];

                if(trans_id==n)
                    dest_id=1;
                else
                    dest_id=trans_id+1;


                for(int tmp=0; tmp<nodenum_of_cell; tmp++)
                {
                    if(cell[i][j].node_in_cell[tmp]==dest_id)
                    {
                        SDtrans(node,time_slot,trans_id,dest_id);

                        if(trans_id==tagged_S)
                          {
                            S_out_number++;
                            SD_num++;
                        }

                        flag=1;
                        break;
                    }
                }

                if(flag==0)
                {

                    /****************transmission scheduling: with probability alpha, do S->R, with probability
1-alpha, do R->D*************/

                    if(probabilityP(alpha))                         //do S->R
                    {
                        int ?_tmp = 0;
                        do {
                            ?_tmp++;
                            index = (int)(nodenum_of_cell * ran0_1(seed));
                            recv_id = cell[i][j].node_in_cell[index];
                            if(node[recv_id].sum_relay_queue_length < sum_relay_queue_max_length && recv_id !=
trans_id) break;
                            if(recv_id == trans_id && ?_tmp == ?)
                                ?_tmp --;
                        } while (?_tmp < ?);

                        SRtrans(node,time_slot,trans_id,recv_id);
                        if(trans_id==tagged_S)
                        {
                            SR_num++;
                            S_out_number++;
                        }

                    }


                    else //do R->D
                    {
                        int ?2_tmp = 0;
                        do {
                            ?2_tmp++;
                            index = (int)(nodenum_of_cell * ran0_1(seed));
                            recv_id = cell[i][j].node_in_cell[index];
```

```
                            int relay_queue_index=0;

                            if(trans_id==n)
                                relay_queue_index=recv_id-1;
                            else if(recv_id<trans_id)
                                relay_queue_index=recv_id;
                            else
                                relay_queue_index=recv_id-2;

                            if(!(node[trans_id].relay_queue[relay_queue_index].empty()) && recv_id != trans_id)
break;

                            if(recv_id == trans_id && ?2_tmp == ?)
                                ?2_tmp --;
                        } while (?2_tmp < ?);

                        RDtrans(node,time_slot,trans_id,recv_id);
                        if(trans_id==tagged_S)
                            RD_num++;
                    }


                }

            }
        }
    }

}
```

SDtrans.cpp

```
#include "my_struc.h"

extern int tagged_S;
extern float delay;
extern float queuing_delay;
extern float delivery_delay;

void SDtrans(struct Node *node, long time_slot, int trans_id, int dest_id)
{
    if(!(node[trans_id].local_queue.empty())) //if the local queue has packet
    {
        node[trans_id].local_queue.front().reception_time=time_slot; //recording the reception time of this
packet

        if(trans_id==tagged_S) //without loss of generality, we focus on a tagged SD pair
        {
            struct Packet packet;
            packet=node[trans_id].local_queue.front();

            queuing_delay=queuing_delay*(1.0*node[dest_id].recv_ct/(node[dest_id].recv_ct+1))+1.0*(packet.
queue_head_time-packet.arrival_time)/(node[dest_id].recv_ct+1);
            delivery_delay=delivery_delay*(1.0*node[dest_id].recv_ct/(node[dest_id].recv_ct+1))+1.0*(packet.
reception_time-packet.queue_head_time)/(node[dest_id].recv_ct+1);
            delay=delay*(1.0*node[dest_id].recv_ct/(node[dest_id].recv_ct+1))+1.0*(packet.reception_time-packet.
arrival_time)/(node[dest_id].recv_ct+1);

        }

        node[trans_id].local_queue.pop();
        node[trans_id].source_queue_length--;
        node[dest_id].recv_ct=node[dest_id].recv_ct+1;

        if(!(node[trans_id].local_queue.empty()))
```

```
        {
            node[trans_id].local_queue.front().queue_head_time=time_slot;

        }


    }
}
```

SRtrans.cpp

```
#include "my_struc.h"

extern int buffer_bound;

//extern int source_queue_max_length;
extern int sum_relay_queue_max_length;

extern int n;
extern int tagged_S;
extern int R_in_number;
extern long lost_number;


void SRtrans(struct Node *node, long time_slot, int trans_id, int recv_id)
{
    int dest_id=0;
    int relay_queue_index=0;
    struct Packet packet;


    /*********if the buffer of relay node is not full!*******************/

    if(node[recv_id].sum_relay_queue_length<sum_relay_queue_max_length)
    {


        if(!node[trans_id].local_queue.empty())
        {
            // the traffic pattern is 1<-->2, 3<-->4, ...
            // compute the destination node id
            if(trans_id==n)
                dest_id=1;
            else
                dest_id=trans_id+1;

            if(recv_id==n)
            {
                relay_queue_index=dest_id-1;
            }
            else if(dest_id<recv_id)
            {
                relay_queue_index=dest_id;
            }
            else
            {
                relay_queue_index=dest_id-2;
            }

            packet=node[trans_id].local_queue.front();
            node[recv_id].relay_queue[relay_queue_index].push(packet);
            node[recv_id].sum_relay_queue_length++;

            if(recv_id==tagged_S)
            {
```

```
                R_in_number++;
            }

            node[trans_id].local_queue.pop();
            node[trans_id].source_queue_length--;

        }
    }

    else
    {
        if(!node[trans_id].local_queue.empty())
        {
            if(trans_id==tagged_S)
            {
                lost_number++;
            }

            if(recv_id==tagged_S)
            {
                R_in_number++;
            }

            node[trans_id].local_queue.pop();
            node[trans_id].source_queue_length--;
        }

    }


    if(!(node[trans_id].local_queue.empty()))
    {
        node[trans_id].local_queue.front().queue_head_time=time_slot;
    }

}
```

RDtrans.cpp

```
#include "my_struc.h"


extern int n;
extern int tagged_D;
extern int R_out_number;
extern float delay;
extern float queuing_delay;
extern float delivery_delay;


void RDtrans(struct Node *node, long time_slot,int trans_id, int recv_id)
{
    int relay_queue_index=0;

    //find the corresponding relay_queue in relay node
    if(trans_id==n)
        relay_queue_index=recv_id-1;
    else if(recv_id<trans_id)
        relay_queue_index=recv_id;
    else
        relay_queue_index=recv_id-2;

    //if relay queue has packet
    if(!(node[trans_id].relay_queue[relay_queue_index].empty()))
    {
```

```
        node[trans_id].relay_queue[relay_queue_index].front().reception_time=time_slot;

        //we only record a node pair
        if(recv_id==tagged_D)
        {
            struct Packet packet;
            packet=node[trans_id].relay_queue[relay_queue_index].front();

            queuing_delay=queuing_delay*(1.0*node[recv_id].recv_ct/(node[recv_id].recv_ct+1))+1.0*(packet.
queue_head_time-packet.arrival_time)/(node[recv_id].recv_ct+1);
            delivery_delay=delivery_delay*(1.0*node[recv_id].recv_ct/(node[recv_id].recv_ct+1))+1.0*(packet.
reception_time-packet.queue_head_time)/(node[recv_id].recv_ct+1);
            delay=delay*(1.0*node[recv_id].recv_ct/(node[recv_id].recv_ct+1))+1.0*(packet.reception_time-packet.
arrival_time)/(node[recv_id].recv_ct+1);

        }

        //testing the output rate of the relay queue
        if(trans_id==tagged_D)
        {
            R_out_number++;
        }

        node[trans_id].relay_queue[relay_queue_index].pop();
        node[trans_id].sum_relay_queue_length--;
        node[recv_id].recv_ct=node[recv_id].recv_ct+1;
    }
}
```