Multimedia Traffic Simulation

Imed CHIHI, Walid ERREBIB, Hassene FARHAT and Zied FAKHFEKH

January 1998

Abstract

This paper[1] presents the results of a multimedia traffic simulation at a router. The policies simulated are First In First Out (FIFO) and Earliest Packet Discard (EPD). We give also the implementation details of our simulator.

1 Introduction

In modern networks, multimedia applications are spreading out quickly. The traffic generated is subject to real time constraints. On the other hand, it is bandwidth-consuming. This study aims to build a simulator to have an insight into a node (router) transferring such traffic. The model studied is a simple network supporting a multimedia traffic with only one path from the source to the destination.

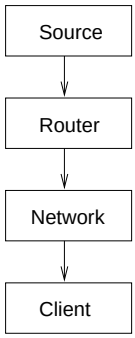


Figure 1. The model simulated. The network is supposed to be a single serial line.

The source throughput is 25 images per second on default which is a reasonable rate for good video rendering quality. An image, part of a video clip, is made up of a certain number of packets. That number is uniformly distributed between 1 and some upper bound (10 on default), a packet is 1024-bit long on default. The generated packets are supposed to reach the router in the same order of their generation. We assumed that the generations dates of the packets of an image are uniformly distributed over the $\frac{1}{25}$th second-long interval.

In the next, we'll present the simulation results of two service disciplines: FIFO and EPD. The FIFO discipline consists into serving the first coming packet first. The EPD requires the definition of a queue size threshold, below that limit the discipline behaves just like FIFO, in converse, beyond the threshold, an incoming image is discarded while packets belonging to already accepted images are queued in.

In section three, we give some implementation details about the simulator and a complete listing.

Section four is an interpretation of the results.

2 Results

We have simulated the behavior of the router over 300 seconds. The network capacity was set to 512000 bps. The other parameters were left set to their default values.

For EPD, the threshold was set to 35 packets.

2.1 Response Time

We will present here the curves of the response time for either FIFO and EPD.

[1] Prepared as a home work of a Simulation course given by Professor Farouk KAMMOUN at the National School for Computer Sciences (ensi), Tunisia.

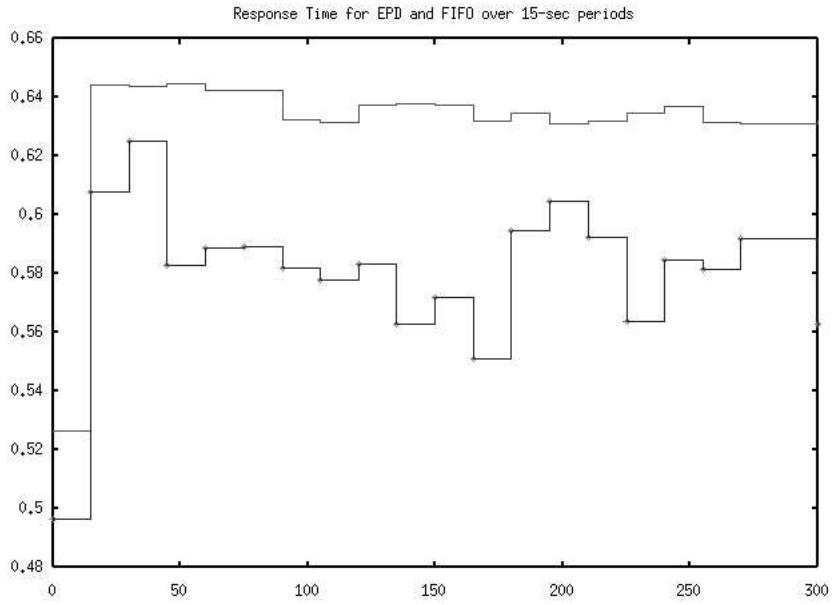


Figure 2. Response time for EPD and FIFO compared. The lines with dots denote EPD.

2.2 Success Rate

The success rate is calculated as the ratio between the number of images that successfully out went the router and the sum of these and those which were rejected. This ratio is registered every time an image is rejected or successfuly routed.

Here is the success rate curves for the two FIFO and EPD.

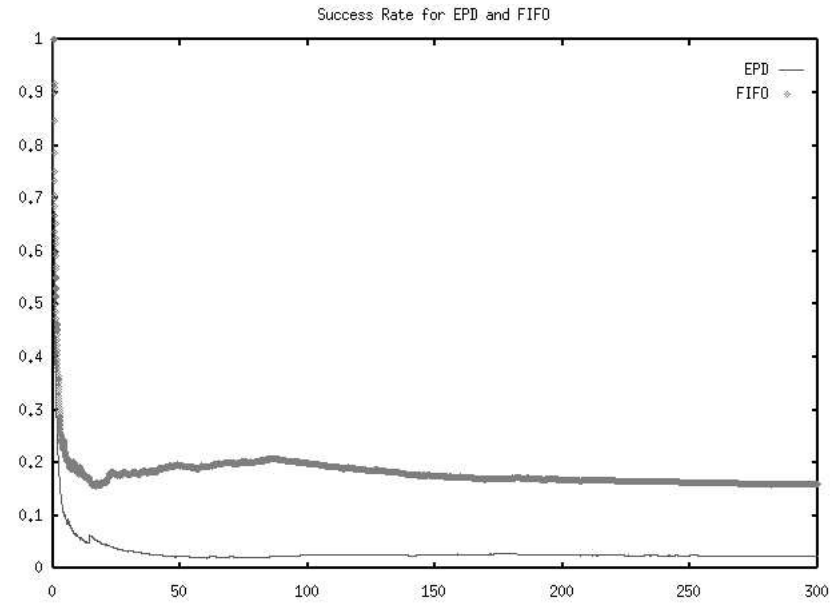
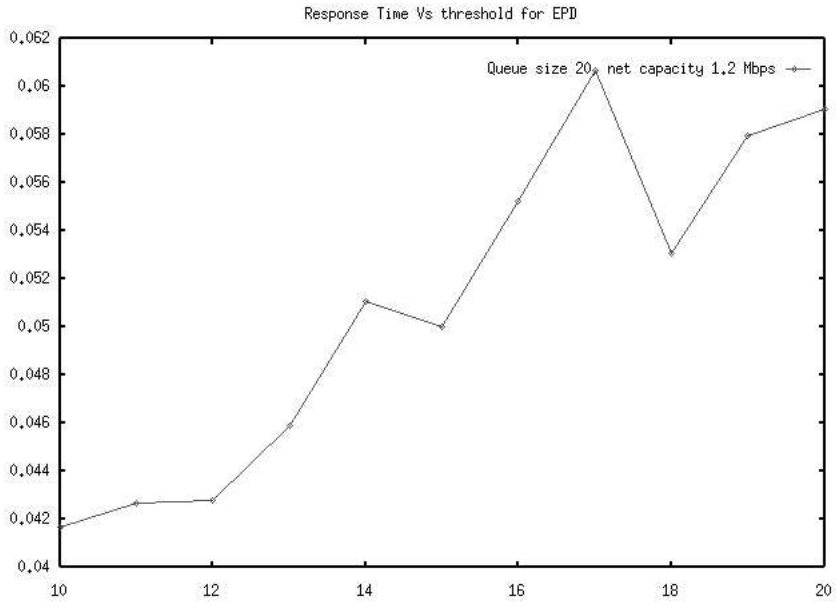


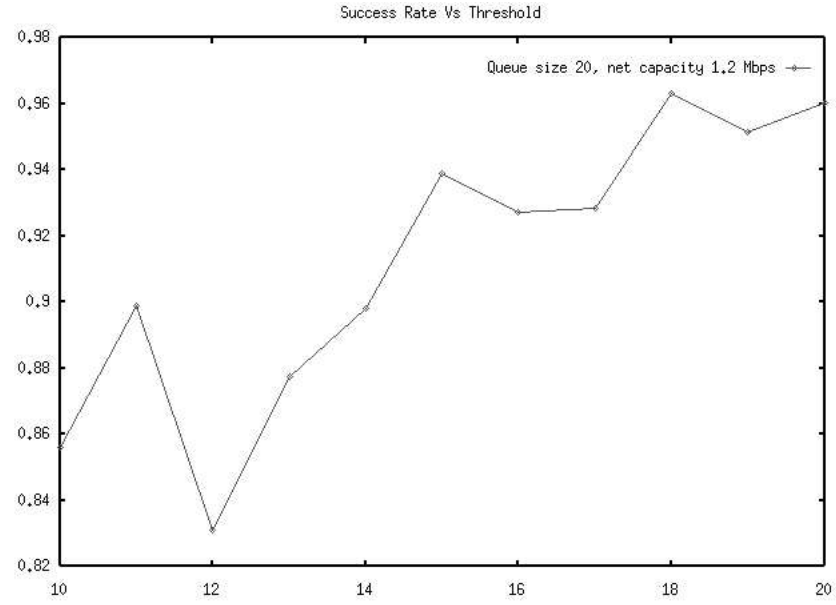
Figure 3. Success rate for FIFO and EPD. The EPD service discipline rejects more packets than FIFO. It appears here with a thin line.

2.3 Threshold Impact

This subsection deals with the sensitivity of the average response time and the average success rate to the threshold (for EPD).



(a)



(b)

Figure 4. The impact of the threshold on the performance of and EPD-based server.

3 Implementation of the Simulator

The simulator was written in C++, and compiled with the GNU C/C++ compiler version 2.7.2.1. We have user i486 and Pentium systems running Linux kernel 2.0.29.

3.1 Classes

We used 9 classes, one of them is virtual (`c_router`). The class hierarchy is outlined in Figure 6.

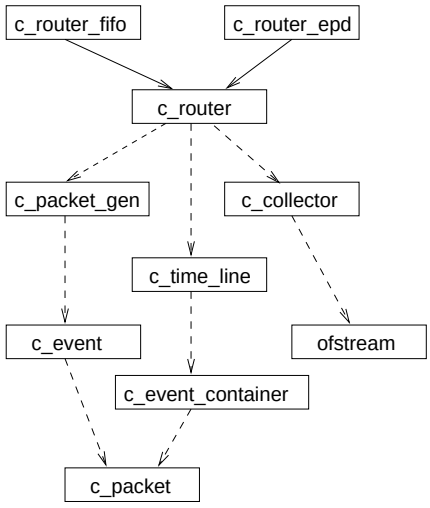


Figure 5. The class hierarchy. Solid lines denote heritage relationships, the dashed lines denote the "uses" relationship.

c_router A virtual class that defines the basic behavior of the simulation kernel.

c_router_fifo Redefines some virtual methods of c_router to fit the FIFO discipline.

c_router_epd Redefines some virtual methods of c_router to fit the EPD discipline.

c_packet_gen Provides the packet generation services with support to random sized images.

c_time_line Handles the queue of the events.

ofstream A standard C++ class providing file output access.

c_event Contains informations on a given event and the packet to which the event is related.

c_packet_container Containes a packet class with a link to another instance of `c_packet_container`. Used in lists of events.

c_packet Contains informations on a given packet and the image to which it's related.

3.2 Compiling and Running the Simulator

The source are made up of five modules, nearly one per class. Modules are compiled using,

```
g++ module_name.cpp -o module_name.o
```

Once all of the five module compiled, one has to build the binary file, using,

```
g++ -o mmsim collector.o event.o packet_gen.o
time_line.o router.o
```

Here is the command line you should.

```
mmsim [-p policy] [-s source_throughput]
[-q packet_size] [-t epd_threshold] [-n
net_capacity] [-d sim_duration] [-c queue_capacity]
```

All the parameters are optional and could be given in any order.

p Sets the service discipline, either FIFO or EPD. Default is FIFO,

s Sets the source throughput inimage per second. Default is 25,

q Sets the packet size in bits. Default is 1024,

t Sets the EPD queue threshold in packets, valid only with EPD service discipline. Default is 20,

n Sets the maximum throughput supported by the network in bits per second. Default is 256000,

d Sets the simulation duration (the time simulated) in seconds. Default is 10,

c Sets the queue capacity in packets. Default is 40.

4 Conclusion

According to the results collected, we can say that when using EPD we gain on response time at the expense of the success rate, as opposed to FIFO.

With EPD, when the threshold increases (up to the whole queue size), both the response time and the success rate increase. When the threshold "approachs" the queue size, EPD tend to behave like FIFO.

After performing many simulations with diffrent input parameters, we can deduce that some stability appears after about 2 seconds.

5 The Source Code

All the include files contain preprocessor primitives to prevent multiple inclusions.

5.1 The `c_event` class

5.1.1 Header

```
#ifndef _EVENT_H
#define _EVENT_H
#include "simul_main.h"

class c_packet
{
  private:
    int image_id; // image identifier
    int packet_id; // packet identifier
    int nb_packets; // number of packets
    double gen_date; // generation date

  public:
    c_packet();
    int set_image_id(int i_id);
    int get_image_id();
    int set_packet_id(int p_id);
    int get_packet_id();
    int set_nb_packets(int nb_p);
    int get_nb_packets();
    int set_gen_date(double g_date);
    double get_gen_date();
    int is_last();
    ~c_packet();
};

class c_event
{
  private:
    int event_type; // event type
    double event_date; // event date

  public:
    c_packet* packet;

    c_event();
    int set_event_type(int e_type);
    int get_event_type();
    int set_event_date(double e_date);
    double get_event_date();
    int set_event_packet(c_packet* pack);
    c_packet* get_event_packet();
    ~c_event();
};

#endif
```

5.1.2 Code

```
#include "event.h"
c_packet::c_packet()
```

```
{
  image_id = 0;
  packet_id = 0;
  nb_packets = 0;
  gen_date = 0.0;
}

int c_packet::set_image_id(int i_id)
{
  image_id = i_id;
  return 0;
}

int c_packet::get_image_id()
{
  return image_id;
}

int c_packet::set_packet_id(int p_id)
{
  packet_id = p_id;
  return OK;
}

int c_packet::get_packet_id()
{
  return packet_id;
}

int c_packet::set_nb_packets(int nb_p)
{
  nb_packets = nb_p;
  return OK;
}

int c_packet::get_nb_packets()
{
  return nb_packets;
}

int c_packet::set_gen_date(double g_date)
{
  gen_date = g_date;
  return OK;
}

double c_packet::get_gen_date()
{
  return gen_date;
}

int c_packet::is_last()
{
  return packet_id == nb_packets - 1;
}

c_packet::~c_packet()
{
```

```
}

c_event::c_event()
{
  packet = new c_packet;
  event_date = 0.0;
  event_type = 0;
}

int c_event::set_event_type(int e_type)
{
  event_type = e_type;
  return OK;
}

int c_event::get_event_type()
{
  return event_type;
}

int c_event::set_event_date(double e_date)
{
  event_date = e_date;
  return OK;
}

double c_event::get_event_date()
{
  return event_date;
}

int c_event::set_event_packet(c_packet* pack)
{
  *packet = *pack;
  return OK;
}

c_packet* c_event::get_event_packet()
{
  return packet;
}

c_event::~c_event()
{
}
```

5.2 The `c_collector` class

5.2.1 Header

```
#ifndef __COLLECTOR_H
#define __COLLECTOR_H

#include "simul_main.h"

class c_collector
{
  private:
    ofstream fd_misc;
```

```
    ofstream fd_rt;
    ofstream fd_sr;
    int safe_images;

  public:
    c_collector();
    int inc_safe_images();
    int get_safe_images();
    int register_rt(double rt, double date, int i_id);
    int register_setting(char *p_name, int p_val);
    int register_success_rate(double sr, double date);
    ~c_collector();
};

#endif
```

5.2.2 Code

```
#include "collector.h"

c_collector::c_collector()
{

  safe_images = 0;

  fd_rt.open("rt.sim");
  if (!fd_rt)
    {
      cerr << "cannot open rt.sim, aborting.." << endl;
      exit(ERR_FILE_SYSTEM);
    }

  fd_misc.open("misc.sim");
  if (!fd_misc)
    {
      cerr << "cannot open misc.sim, aborting.." << endl;
      exit(ERR_FILE_SYSTEM);
    }

  fd_sr.open("sr.sim");
  if (!fd_sr)
    {
      cerr << "cannot open sr.sim, aborting.." << endl;
      exit(ERR_FILE_SYSTEM);
    }
}

int c_collector::inc_safe_images()
{
  safe_images++;
  return OK;
}

int c_collector::get_safe_images()
{
  return safe_images;
}
```

```
int c_collector::register_rt(double rt, double date, int i_id)
{
  fd_rt << date << "\t" << rt << "\t" << i_id << endl;
  return OK;
}

int c_collector::register_setting(char *p_name, int p_val)
{
  fd_misc << p_name << "\t\t" << p_val << endl;
  return OK;
}

int c_collector::register_success_rate(double sr, double date)
{
  fd_sr << date << "\t" << sr << endl;
  return OK;
}

c_collector::~c_collector()
{
  fd_rt.close();
  fd_misc.close();
}
```

5.3 The c_packet_gen class

5.3.1 Header

```
#ifndef __PACKET_GEN_H
#define __PACKET_GEN_H

#include "simul_main.h"
#include "event.h"

class c_packet_generator
{
  public:
    int image_seq, packet_seq;
    int current_image_size_in_packets, last_packet_id;
    int isize_lower_bound, isize_upper_bound;
    int packet_size;
    int source_throughput;

  public:
    c_packet_generator(int lower_bound, int upper_bound, int p_size, int
s_throughput);
    c_packet_generator();

  //private:
    int gen_image_size();

  public:
    int gen_packet(c_packet *pack, double c_date);
    int get_source_throughput();
    int get_packet_size();
    int get_image_count();
    ~c_packet_generator();
};
```

```
#endif
```

5.3.2 Code

```
#include "packet_gen.h"

c_packet_generator::c_packet_generator(int lower_bound, int upper_bound, int
p_size, int s_throughput)
{
  image_seq = 0;
  packet_seq = 0;
  current_image_size_in_packets = 1;
  last_packet_id = 0;
  isize_lower_bound = lower_bound;
  isize_upper_bound = upper_bound;
  packet_size = p_size;
  source_throughput = s_throughput;
  srand((int) time(NULL));
}

c_packet_generator::c_packet_generator()
{
  image_seq = 0;
  packet_seq = 0;
  current_image_size_in_packets = 1;
  last_packet_id = 0;
  isize_lower_bound = DEFAULT_LOWER_BOUND;
  isize_upper_bound = DEFAULT_UPPER_BOUND;
  packet_size = DEFAULT_PACKET_SIZE;
  source_throughput = DEFAULT_SOURCE_THROUGHPUT;
  srand((int) time(NULL));
}

int c_packet_generator::gen_image_size()
{
     // generates a number between isize_lower_bound
     // and isize_upper_bound
   return (int)(isize_lower_bound +
               (isize_upper_bound - isize_lower_bound)*(rand()/(1.0*RAND_MAX)));
}


int c_packet_generator::gen_packet(c_packet *pack, double c_date)
{
  ASSERT_VALID(pack);

     // is this the first packet, or a part of an image
     // being processed ?
  if (last_packet_id == current_image_size_in_packets - 1)
    {
      current_image_size_in_packets = (int) (gen_image_size()/(1.0*packet_size))
+ 1;
      image_seq++;
      packet_seq=0;
    }
  else
    {
      packet_seq++;
```

```
    }

  last_packet_id = packet_seq;

  pack->set_image_id(image_seq);
  pack->set_packet_id(packet_seq);
  pack->set_nb_packets(current_
image_size_in_packets);
  pack->set_gen_date(c_date);

  return 0;
}

int c_packet_generator::get_source_throughput()
{
  return source_throughput;
}

int c_packet_generator::get_packet_size()
{
  return packet_size;
}

int c_packet_generator::get_image_count()
{
  return image_seq;
}

c_packet_generator::~c_packet_generator()
{
}
```

5.4 The `c_router` class

5.4.1 Header

```
#ifndef __ROUTER_H
#define __ROUTER_H

#include "collector.h"
#include "time_line.h"
#include "packet_gen.h"

double current_date = 0.0; //the current date
double sim_duration = 0.0; //duration of simulation

class c_router :  c_time_line
{
  public:
    c_packet_generator *packet_gen;
    c_collector collector;
    c_time_line *time_line;
    double last_departure;
    double current_date, simulation_date;
    int queue_capacity;
    int queue_size;

  private:
    int net_capacity;           // in bps
```

```
  public:
    c_router ();
    c_router(int net_cap, double sd, int queue_cap);
    int get_net_capacity();
    int init_simulation ();
    virtual int start_simulation ();
    int stop_simulation ();
    ~c_router ();
};

class c_router_fifo :  public c_router
{
  public:
    c_router_fifo(); // deault constructor
    c_router_fifo(int lb, int ub, int st, int p_size, int net_cap, double sd,
int qc);
    int start_simulation();
};

class c_router_epd :  public c_router
{
  private:
  int epd_threshold;

  public:
    c_router_epd(); // deault constructor
    c_router_epd(int lb, int ub, int st, int p_size, int tepd, int net_cap,
double sd, int qc);
    int get_epd_threshold();
    int start_simulation();
};

#endif
```

5.4.2 Code

```
#include "router.h"

c_router::c_router()
{
  current_date = 0.0;
  sim_duration = DEFAULT_SIM_DURATION;
  last_departure = 0.0;
  net_capacity = DEFAULT_NET_CAPACITY;
  queue_capacity = DEFAULT_QUEUE_CAPACITY;
  time_line = new c_time_line;
  queue_size = 0;
}

c_router::c_router(int net_cap, double sd, int queue_cap)
{
  current_date = 0.0;
  sim_duration = sd;
  last_departure = 0.0;
  net_capacity = net_cap;
  queue_capacity = queue_cap;
  time_line = new c_time_line;
```

```
  queue_size = 0;
}

int c_router::get_net_capacity()
{
  return net_capacity;
}

int c_router::init_simulation()
{
  c_event first_event;

  packet_gen->gen_packet(first_event.packet, current_date);
  cout << "First event generated" << endl;                    //-

  time_line->insert_event(EVENT_ARRIVAL, current_date, first_event.packet);
  cout << "First event inserted" << endl;                     //-

  return OK;
}

int c_router::start_simulation()
{
  return OK;
}

c_router::~c_router()
{
}

c_router_fifo::c_router_fifo() :  c_router()
{
  packet_gen = new c_packet_generator();
}

c_router_fifo::c_router_fifo(int lb, int ub, int st, int p_size, int net_cap,
double sd, int qc) :  c_router(net_cap, sd, qc)
{
  packet_gen = new c_packet_generator(lb, ub, p_size, st);
}

int c_router_fifo::start_simulation ()
{
  c_event current_event, new_event;
  int rejected_images = 0;

  int discard_flag = FALSE, success_flag = TRUE;
  int last_successful_packet = -1;
  int current_image = 1;
  int this_image;
  int current_image_size = 0;

    // time between the generation of two consecutive packets
  double pack_intergen;
  double service_time;

  collector.register_setting("policy (FIFO)", POLICY_FIFO);
  collector.register_setting("packet_size", packet_gen->get_packet_size());
```

```
  collector.register_setting("source_
throughput", packet_gen->get_source_throughput());
  collector.register_setting("net_capacity", get_net_
capacity());

  while (current_date < sim_duration)
    {
      // if there are no more events, exit simulation
      if (time_line->is_empty ())
        return DO_NOT_MIND;

      // pick up the first (earliest) event on the time_line
      // this event is automatically discarded
      current_event = time_line->retrieve_event ();
      current_date = current_event.get_event_date();

      cout << "current date = " << current_date << " qsize = " <<
queue_size << endl;   //-

      switch (current_event.get_event_type ())
        {
        case EVENT_ARRIVAL:
          cout << "start_simulation():  processing EVENT_ARRIVAL" << endl;

          // generating one arrival
          if (queue_size >= queue_capacity)
            {
              discard_flag = TRUE;
              if (current_event.packet->is_last())
                {
                  rejected_images++;
                  discard_flag = FALSE;
                  collector.register_success_rate(collector.get_safe_images()/
(1.0*(collector.get_safe_images()+rejected_images)), current_date);
                }
              packet_gen->gen_packet(new_event.packet, current_date);
              pack_intergen =
packet_gen->get_source_throughput()*current_event.packet->get_nb_packets();
              time_line->insert_event(EVENT_ARRIVAL, current_date +
1/pack_intergen, new_event.packet);
              break;
            }

          packet_gen->gen_packet(new_event.packet, current_date);
          pack_intergen =
packet_gen->get_source_throughput()*current_event.packet->get_nb_packets();
          time_line->insert_event(EVENT_ARRIVAL, current_date + 1/pack_intergen,
new_event.packet);
          queue_size++;

          // generating one entry
          service_time =
8*packet_gen->get_packet_size()/(1.0*get_net_capacity());
          if (current_date < last_departure)
            {
              time_line->insert_event(EVENT_ENTRY, last_departure,
current_event.packet);
              last_departure += service_time;
```

```
            }
          else
            {
              time_line->insert_event(EVENT_ENTRY, current_date,
current_event.packet);
              last_departure = current_date + service_time;
            }

          if (current_event.packet->is_last() && discard_flag)
            {
              rejected_images++;
              discard_flag = FALSE;
              collector.register_success_rate((int)collector.get_safe_images()/
(1.0*(collector.get_safe_images()+rejected_images)),
current_date);
            }
          break;

        case EVENT_ENTRY:
          cout << "start_simulation():  processing EVENT_ENTRY" << endl;
          service_time =
8*packet_gen->get_packet_size()/(1.0*get_net_capacity());
          time_line->insert_event(EVENT_DEPARTURE, current_date + service_time,
current_event.packet);
          break;

        case EVENT_DEPARTURE:
          cout << "start_simulation():  processing EVENT_DEPARTURE >> " <<
current_event.packet->get_image_id() << "-" <<
current_event.packet->get_packet_id() << endl;
          queue_size--;

          // read the outgoing packet's image id
          this_image = current_event.packet->get_image_id();

          // does it belong to a new image ?
          if ((this_image != current_image) || (current_image == 1))
            {
              if ((current_image_size == last_successful_packet + 1) &&
success_flag)
                {
                  // gather stats on success rate
                  pack_intergen =
1/(packet_gen->get_source_throughput()*current_event.packet->get_nb_packets());
                  collector.register_rt(current_date -
(current_event.packet->get_gen_date() -
pack_intergen*(current_event.packet->get_nb_packets() - 1)), current_date,
/*current_event.packet->get_image_id()*/ current_image);
                  collector.inc_safe_images();
                  collector.register_success_rate((double)collector.get_safe_images()/
(1.0*(collector.get_safe_images()+rejected_images)), current_date);
                }

              // reset reject detection settings
              success_flag = TRUE;
              last_successful_packet = -1;
              current_image = this_image;
              current_image_size = current_event.packet->get_nb_packets();
```

```
            }

          // have we lost the expected packet ?
          // if yes, unset the success flag.  That is, declare the image lost
          if (current_event.packet->get_packet_id() != last_successful_packet +
1)
            success_flag = FALSE;

          // set the expected flag
          last_successful_packet = current_event.packet->get_packet_id();
          break;
        }
    }

  collector.register_setting("safe_images", collector.get_safe_images());
  collector.register_setting("generated_images", packet_gen->get_image_count());

  return OK;
}

c_router_fifo::~c_router_fifo()
{
}

c_router_epd::c_router_epd() :  c_router()
{
  epd_threshold = DEFAULT_EPD_THRESHOLD;
}

c_router_epd::c_router_epd(int lb, int ub, int st, int p_size, int tepd, int
net_cap, double sd, int qc) :  c_router(net_cap, sd, qc)
{
  epd_threshold = tepd;
  packet_gen = new c_packet_generator(lb, ub, p_size, st);
}

int c_router_epd::get_epd_threshold()
{
  return epd_threshold;
}

int c_router_epd::start_simulation ()
{
  c_event current_event, new_event;
  int rejected_images = 0;

  int discard_image_flag = FALSE, discard_flag = FALSE, success_flag = TRUE;
  int last_successful_packet = -1;
  int current_image = 1, c_image = 1;
  int this_image, t_image;
  int current_image_size = -1;

    // time between the generation of two consecutive packets
  double pack_intergen;
  double service_time;

  collector.register_setting("policy (EPD)", POLICY_EPD);
  collector.register_setting("packet_size", packet_gen->get_packet_size());
```

```
  collector.register_setting("source_throughput",
packet_gen->get_source_throughput());
  collector.register_setting("EPD_threshold", epd_threshold);
  collector.register_setting("net_capacity", get_net_capacity());

  while (current_date < sim_duration)
    {
      // if there are no more events, exit simulation
      if (time_line->is_empty ())
        return DO_NOT_MIND;


      // pick up the first (earliest) event on the time_line
      // this event is automatically discarded
      current_event = time_line->retrieve_event ();
      current_date = current_event.get_event_date();

      cout << "current date = " << current_date << " qsize = " <<
queue_size << "   packet_id = " << current_event.packet->get_packet_id() <<
endl;     //-

      switch (current_event.get_event_type ())
        {
        case EVENT_ARRIVAL:
          cout << "start_simulation():  processing EVENT_ARRIVAL" << endl;

          // generating one arrival
          packet_gen->gen_packet(new_event.packet, current_date);
          pack_intergen =
packet_gen->get_source_throughput()*current_event.packet->get_nb_packets();
          time_line->insert_event(EVENT_ARRIVAL, current_date + 1/pack_intergen,
new_event.packet);

          if (queue_size >= queue_capacity)
            {
              cout << "Rejecting ..." << endl;
              discard_flag = TRUE;
              if (current_event.packet->is_last())
                {
                  rejected_images++;
                  discard_flag = FALSE;
                  discard_image_flag = FALSE;
                  collector.register_success_rate(collector.get_safe_images()/
(1.0*(collector.get_safe_images()+rejected_images)), current_date);
                }
              break;
            }

          t_image = current_event.packet->get_image_id();
          if (queue_size >= epd_threshold)
            {
              cout << "q:  beyond the threshold" << endl; //-
              if (t_image != c_image)
                {
                  if (discard_image_flag)
                    {
                      rejected_images++;
                      collector.register_success_rate(collector.get_safe_images()/
```

```
(1.0*(collector.get_safe_images()+rejected_images)), current_date);
                    }
                  discard_image_flag = TRUE;
                  c_image = t_image;
                  break;
                }
            }
          c_image = t_image;
          queue_size++;

          // generating one entry
          service_time =
8*packet_gen->get_packet_size()/(1.0*get_net_capacity());
          if (current_date < last_departure)
            {
              time_line->insert_event(EVENT_ENTRY, last_departure,
current_event.packet);
              last_departure += service_time;
            }
          else
            {
              time_line->insert_event(EVENT_ENTRY, current_date,
current_event.packet);
              last_departure = current_date + service_time;
            }

          if (current_event.packet->is_last() && discard_flag)
            {
              rejected_images++;
              discard_flag = FALSE;
              discard_image_flag = FALSE;
              collector.register_success_rate((int)collector.get_safe_images()/
(1.0*(collector.get_safe_images()+rejected_images)), current_date);
            }
          break;

        case EVENT_ENTRY:
          cout << "start_simulation():  processing EVENT_ENTRY" << endl;
          service_time =
8*packet_gen->get_packet_size()/(1.0*get_net_capacity());
          time_line->insert_event(EVENT_DEPARTURE, current_date + service_time,
current_event.packet);
          break;

        case EVENT_DEPARTURE:
          cout << "start_simulation():  processing EVENT_DEPARTURE >> " <<
current_event.packet->get_image_id() << "-" <<
current_event.packet->get_packet_id() << endl;
          queue_size--;

          // read the outgoing packet's image id
          this_image = current_event.packet->get_image_id();

          // does it belong to a new image ?
          if ((this_image != current_image) || (current_image == 1))
            {
              if ((current_image_size == last_successful_packet + 1) &&
success_flag)
```

```
                {
                  // gather stats on success rate
                  pack_intergen =
1/(packet_gen->get_source_throughput()*current_event.packet->get_nb_packets());
                  collector.register_rt(current_date -
(current_event.packet->get_gen_date() -
pack_intergen*(current_event.packet->get_nb_packets() - 1)), current_date,
current_image);
                  collector.inc_safe_images();
                  collector.register_success_rate((double)collector.get_safe_images()/
(1.0*(collector.get_safe_images()+rejected_images)), current_date);
                }

              // reset reject detection settings
              success_flag = TRUE;
              last_successful_packet = -1;
              current_image = this_image;
              current_image_size = current_event.packet->get_nb_packets();
            }

          // have we lost the expected packet ?
          // if yes, unset the success flag.  That is, declare the image lost
          if (current_event.packet->get_packet_id() != last_successful_packet +
1)
            success_flag = FALSE;

          last_successful_packet = current_event.packet->get_packet_id();
          break;
        }
    }

  collector.register_setting("safe_images", collector.get_safe_images());
  collector.register_setting("generated_images", packet_gen->get_image_count());

  return OK;
}

c_router_epd::~c_router_epd()
{
}

int main(int argc, char **argv)
{
  if (argc == 1)
    {
      c_router_fifo router;
      router.init_simulation();
      router.start_simulation();
      return 0;
    }

  int p_policy             = DEFAULT_POLICY;
  int p_source_throughput  = DEFAULT_SOURCE_THROUGHPUT;
  int p_packet_size        = DEFAULT_PACKET_SIZE;
  int p_epd_threshold      = DEFAULT_EPD_THRESHOLD;
  int p_net_capacity       = DEFAULT_NET_CAPACITY;
  int p_queue_capacity     = DEFAULT_QUEUE_CAPACITY;
  int p_sim_duration       = DEFAULT_SIM_DURATION;
```

```
  int p_parser = 1;
  while(p_parser < argc)
  {
  switch (argv[p_parser][1])
    {
      case 'p' :
        if (argv[p_parser+1][0]=='E' || argv[p_parser+1][0]=='e')
                  p_policy = POLICY_EPD;
        p_parser++;
        break;

      case 's' :
        sscanf(argv[p_parser+1], "%d", &p_source_throughput);
        p_parser++;
        break;

      case 'q' :
        sscanf(argv[p_parser+1], "%d", &p_packet_size);
        p_parser++;
        break;

      case 't' :
        sscanf(argv[p_parser+1], "%d", &p_epd_threshold);
        p_parser++;
        break;

      case 'n' :
        sscanf(argv[p_parser+1], "%d", &p_net_capacity);
        p_parser++;
        break;

      case 'd' :
        sscanf(argv[p_parser+1], "%d", &p_sim_duration);
        p_parser++;
        break;

      case 'c' :
        sscanf(argv[p_parser+1], "%d", &p_queue_capacity);
        p_parser++;
        break;

      case '-' :
        cout << "Usage:  " << argv[0] << " [-p policy] [-s
source_throughput] [-q packet_size] [-t epd_threshold] [-n net_capacity] [-d
sim_duration] [-c queue_capacity]" << endl <<
        "p      FIFO or EPD (defaults to FIFO)" << endl <<
        "s      in images per second (defaults to " <<
DEFAULT_SOURCE_THROUGHPUT << ")" << endl <<
        "q      in bits (defaults to " << DEFAULT_PACKET_SIZE << ")" << endl
<<
        "t      in packets (defaults to " << DEFAULT_EPD_THRESHOLD << ")" <<
endl <<
        "n      in packets (defaults to " << DEFAULT_NET_CAPACITY << ")"<<
endl <<
        "d      in bits per second (defaults to " << DEFAULT_SIM_DURATION <<
")" << endl <<
```

```
        "c      in packets (defaults to " << DEFAULT_QUEUE_CAPACITY << ")"
<< endl;
          p_parser++;
        return 0;

      default:
        p_parser++;
    }
    }

  if (p_policy == POLICY_FIFO)
    {
      c_router_fifo router(0, 1024*10-1, p_source_throughput, p_packet_size,
p_net_capacity, p_sim_duration, p_queue_capacity);
      router.init_simulation();
      router.start_simulation();
    }
  else
    {
      c_router_epd router(0, 1024*10-1, p_source_throughput, p_packet_size,
p_epd_threshold, p_net_capacity, p_sim_duration, p_queue_capacity);
      router.init_simulation();
      router.start_simulation();
    }

  return 0;
}
```

5.5 The c_time_line class

5.5.1 Header

```
#ifndef __TIME_LINE_H
#define __TIME_LINE_H

#include "simul_main.h"
#include "event.h"

class c_event_container
{
  public:
    c_event event;
    c_event_container* link;

    c_event_container();
    ~c_event_container();
};


class c_time_line
{
  private:
    c_event_container* top_events;

  public:
    time_line();
    int is_empty();
    int insert_event(int e_type, double e_date, c_packet* e_packet);
    c_event retrieve_event();
```

```
    int remove_event(c_event *event);
    ~c_time_line();
};

#endif
```

5.5.2 Code

```
#include "time_line.h"

c_event_container::c_event_container()
{
  link = NULL;
};

c_event_container::~c_event_container()
{
};

c_time_line::c_time_line()
{
  top_events = NULL;
}

int c_time_line::is_empty()
{
  return (top_events == NULL);
}

int c_time_line::insert_event(int e_type, double e_date, c_packet* e_packet)
{
  if (is_empty())
    {
      top_events = new c_event_container;
      top_events->event.set_event_type(e_type);
      top_events->event.set_event_date(e_date);
      top_events->event.set_event_packet(e_packet);
      return OK;
    }

  c_event_container *traveller_ptr = top_events;

  if (top_events->event.get_event_date() >= e_date)
    {
      top_events = new c_event_container;
      top_events->event.set_event_type(e_type);
      top_events->event.set_event_date(e_date);
      top_events->event.set_event_packet(e_packet);
      top_events->link = traveller_ptr;
      return OK;
    }

  while (traveller_ptr->link != NULL)
    {
      if (traveller_ptr->link->event.get_event_date() < e_date)
      {
        traveller_ptr = traveller_ptr->link;
      }
```

```
      else
        break;
    }

  c_event_container *t_ptr = new c_event_container;
  t_ptr->event.set_event_type(e_type);
  t_ptr->event.set_event_date(e_date);
  t_ptr->event.set_event_packet(e_packet);
  t_ptr->link = traveller_ptr->link;
  traveller_ptr->link = t_ptr;
  return OK;
}

c_event c_time_line::retrieve_event()
{
  c_event ret_event;
  c_event_container* tptr = top_events;

  ret_event = top_events->event;

  top_events = top_events->link;
  delete tptr;

  return ret_event;
}

int c_time_line::remove_event(c_event* event)
{
  c_event_container *traveller_ptr;
  c_event_container *tptr;
  if (&(top_events->event) == event)
    {
      traveller_ptr = top_events;
      top_events = top_events->link;
      delete traveller_ptr;
      return OK;
    }

  traveller_ptr = top_events;
  while (&(traveller_ptr->link->event) != event)
    {
      traveller_ptr = traveller_ptr->link;
      if (traveller_ptr->link == NULL)
        return DO_NOT_MIND;
    }

  tptr = traveller_ptr->link;
  traveller_ptr->link = traveller_ptr->link->link;
  delete tptr;
  return OK;
}

c_time_line::~c_time_line()
{
}
```

5.6 The common header simul_main.h

```
#ifndef _SIMUL_MAIN_H
#define _SIMUL_MAIN_H

#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>
#include <time.h>       // to set the seed of the generator

  // makes sure the pointer on the object passed as a parameter
  // is not null
#define ASSERT_VALID(ptr) if (ptr==NULL) return 2

#define DEFAULT_PACKET_SIZE       1024
#define DEFAULT_LOWER_BOUND       0
#define DEFAULT_UPPER_BOUND       DEFAULT_PACKET_SIZE * 10 - 1;
#define DEFAULT_SOURCE_THROUGHPUT 25
#define DEFAULT_EPD_THRESHOLD     20
#define DEFAULT_NET_CAPACITY      256000
#define DEFAULT_QUEUE_CAPACITY    40
#define DEFAULT_SIM_DURATION      10

#define POLICY_FIFO               0
#define POLICY_EPD                1
#define DEFAULT_POLICY            POLICY_FIFO

#define EVENT_ARRIVAL             0
#define EVENT_ENTRY               1
#define EVENT_DEPARTURE           2

#define ERR_FILE_SYSTEM 10
#define OK               0
#define DO_NOT_MIND     1

#define FALSE           0
#define TRUE            1

extern double current_date;
extern double sim_duration;

#endif
```