Networks Lab Endsem

106119029, Dipesh Kafle

Contents

Question 1

Code

- server.cpp

```
#include <arpa/inet.h>
#include <functional>
#include <iostream>
#include <netdb.h>
#include <netinet/in.h>
#include <optional>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <thread>
#include <unistd.h>
#include <unordered_map>
#include <vector>

using namespace std;
```

```
void send_int(int fd, int val) {
  val = htonl(val);
  send(fd, &val, sizeof(int), 0);
}
int recv_int(int fd) {
  int x;
  recv(fd, &x, sizeof(int), 0);
  return ntohl(x);
}

void HANDLE_SEND_RECV_ERRORS(int st) {
  if (st == -1) {
    perror("Error in send()/recv()");
    exit(254);
  } else if ((st) == 0) {
    perror("Connection is closed because send/recv returned 0");
    exit(255);
  }
}

void die_with_error(const char *message, int err_code = 1) {
  perror(message);
  exit(err_code);
}

struct Server {
  int sock_fd;
  struct addrinfo *result;
  static unordered_map<string, int> ip_to_fd;

  Server(const char *address, const char *port_no) {
    this->result = Server::addr_setup(address, port_no);
    this->sock_fd = Server::setup_socket(this->result);
  }

  ~Server() {
    freeaddrinfo(this->result);
    close(sock_fd);
  }

  int listen(int n) { return ::listen(sock_fd, n); }

  std::pair<int, struct sockaddr_storage> accept_connection() {
    struct sockaddr_storage addr;
    socklen_t len = sizeof(addr);
    int client_fd = accept(this->sock_fd, (struct sockaddr *)&addr, &len);
    if (client_fd == -1)
      die_with_error("accept()");
    return {client_fd, addr};
  }

  static void handle_client(int client_fd,
                            const struct sockaddr_storage &addr) {
    string my_ip = Server::get_ip_port(&addr).first + "/" +
                   to_string(Server::get_ip_port(&addr).second);
    while (true) {
      int op = recv_int(client_fd);
      if (op == 0) {
        ip_to_fd.erase(my_ip);
        break;
      } else if (op == 1) {
        string s;
        for (auto &[k, v] : ip_to_fd) {
          s += k;
          s += '\n';
        }
        send_int(client_fd, s.size());
        Server::send<string>(client_fd, s, [](string t) {
          vector<uint8_t> vec;
          copy(t.begin(), t.end(), back_inserter(vec));
          return vec;
        });
      } else if (op == 2) {
        int msg_size = recv_int(client_fd);
        string msg = Server::receive<string>(
            client_fd, msg_size, [](const char *buf) { return buf; });

        int ip_sz = recv_int(client_fd);

        string dest_ip = Server::receive<string>(
            client_fd, ip_sz, [](const char *buf) { return buf; });

        // sending message to the dest
        msg += "\n\n by IP: " + my_ip + "\n";
        send_int(ip_to_fd[dest_ip], msg.size());
        Server::send<string>(client_fd, msg, [](string t) {
          vector<uint8_t> vec;
          copy(t.begin(), t.end(), back_inserter(vec));
          return vec;
        });
      }
    }
```

```
}

static std::pair<std::string, int> get_ip_port(const sockaddr_storage *addr) {
  int port;
  char buf[1000];
  if (addr->ss_family == AF_INET) {
    port = ((struct sockaddr_in *)addr)->sin_port;
    inet_ntop(addr->ss_family, &((struct sockaddr_in *)addr)->sin_addr, buf,
              sizeof(sockaddr_storage));
  } else {
    port = ((struct sockaddr_in6 *)addr)->sin6_port;
    inet_ntop(addr->ss_family, &((struct sockaddr_in6 *)addr)->sin6_addr, buf,
              sizeof(sockaddr_storage));
  }
  return {std::string(buf), port};
}

template <typename T>
static void send(int client_fd, T message,
                 std::function<vector<uint8_t>(T)> f) {
  //
  auto data = f(move(message));
  auto st = ::send(client_fd, &data[0], data.size(), 0);
  // HANDLE_SEND_RECV_ERRORS(st);
}
template <typename T>
static T receive(int client_fd, size_t sz, std::function<T(const char *)> f) {
  //
  char buf[sz + 10];
  auto st = recv(client_fd, buf, sz, 0);
  // HANDLE_SEND_RECV_ERRORS(st);
  buf[st] = 0;
  return f(buf);
}

static int setup_socket(struct addrinfo *result) {
  int yes = 1;
  int sock_fd =
      socket(result->ai_family, result->ai_socktype, result->ai_protocol);
  if (sock_fd < 0)
    die_with_error("socket()");
  if (setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT | SO_LINGER,
                 &yes, sizeof(yes)) < 0)
    die_with_error("setsockopt");
  if (bind(sock_fd, result->ai_addr, result->ai_addrlen) < 0)
    die_with_error("bind()");
    return sock_fd;
  }
  static struct addrinfo *addr_setup(const char *address, const char *port_no) {
    struct addrinfo hints, *res;

    memset(&hints, 0, sizeof(hints));
    hints.ai_addr = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    int status = getaddrinfo(address, port_no, &hints, &res);
    if (status != 0)
      die_with_error("getaddrinfo");
    return res;
  }
};
unordered_map<string, int> Server::ip_to_fd = unordered_map<string, int>();

int main() {
  string port;
  cout << "enter port: ";
  cin >> port;
  Server s(NULL, port.c_str());
  s.listen(10);
  cout << "Here\n";

  vector<std::pair<int, struct sockaddr_storage>> conns;
  while (true) {
    auto [conn, addr] = s.accept_connection();
    cout << "Connected to : " << Server::get_ip_port(&addr).first << '/'
         << Server::get_ip_port(&addr).second << '\n';
    conns.push_back({conn, addr});
    Server::ip_to_fd[Server::get_ip_port(&addr).first + "/" +
                     to_string(Server::get_ip_port(&addr).second)] = conn;
    thread t(Server::handle_client, conns.back().first,
             std::ref(conns.back().second));
    t.detach();
  }
}
```

- client.cpp

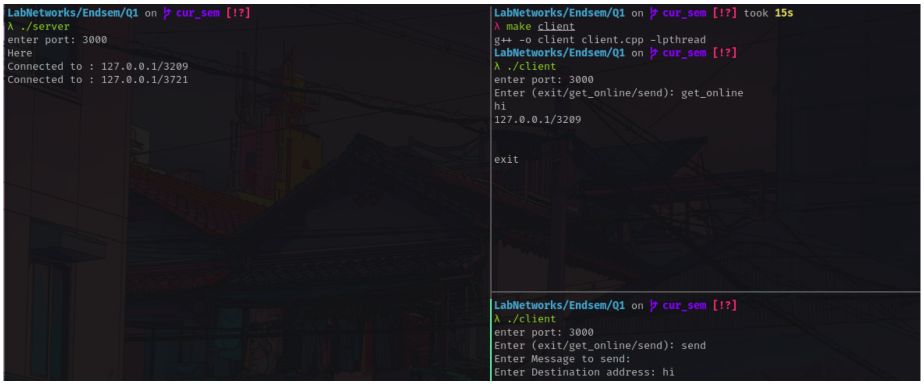
```
#include <arpa/inet.h>
#include <functional>
#include <iostream>
#include <netdb.h>
#include <netinet/in.h>
```

```cpp
#include <optional>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <thread>
#include <unistd.h>
#include <vector>

using namespace std;

void HANDLE_SEND_RECV_ERRORS(int st) {
  if (st == -1) {
    perror("Error in send()/recv()");
    exit(254);
  } else if ((st) == 0) {
    perror("Connection is closed because send/recv returned 0");
    exit(255);
  }
}

void die_with_error(const char *message, int err_code = 1) {
  perror(message);
  exit(err_code);
}

void send_int(int fd, int val) {
  val = htonl(val);
  send(fd, &val, sizeof(int), 0);
}
int recv_int(int fd) {
  int x;
  recv(fd, &x, sizeof(int), 0);
  return ntohl(x);
}

struct Client {
  int client_fd;
  struct addrinfo *result;

  Client(const char *address, const char *port_no) {
    this->result = Client::addr_setup(address, port_no);
    this->client_fd = Client::setup_client_socket(this->result);
  }
```

```
~Client() {
  freeaddrinfo(result);
  close(client_fd);
}

void receiver_thread() {
  while (true) {
    int msg_size = recv_int(this->client_fd);
    auto messsage = this->receive<string>(this->client_fd, msg_size,
                                          [](const char *s) { return s; });
    cout << messsage << '\n';
    cout << '\n';
  }
}
void handle_connection() {
  // Handle the connection
  thread t([&]() { this->receiver_thread(); });
  t.detach();

  while (true) {
    cout << "Enter (exit/get_online/send): ";
    string s;
    cin >> s;
    if (s == "exit") {
      int zero = 0;
      send_int(this->client_fd, 0);
      break;
    } else if (s == "get_online") {
      send_int(this->client_fd, 1);
      int sz = recv_int(this->client_fd);
      auto messsage = this->receive<string>(this->client_fd, sz,
                                            [](const char *s) { return s; });
      cout << "All Online are: \n";
      cout << messsage << '\n';
      cout << '\n';
      cout << '\n';
      cout << '\n';
      // this->receive(this->client_fd, size_t sz,
      // std::function<T(const char *)> f)

    } else if (s == "send") {
      send_int(this->client_fd, 2);
      cout << "Enter Message to send: ";
      string s, dest;
      std::getline(std::cin, s);
      cout << '\n';
      cout << "Enter Destination address: ";
      std::getline(std::cin, dest);
      cout << '\n';
      cout << '\n';
      cout << '\n';
      send_int(this->client_fd, s.size());
      this->send<string>(this->client_fd, s, [](string t) {
        vector<uint8_t> vec;
        copy(t.begin(), t.end(), back_inserter(vec));
        return vec;
      });
      send_int(this->client_fd, dest.size());
      this->send<string>(this->client_fd, dest, [](string t) {
        vector<uint8_t> vec;
        copy(t.begin(), t.end(), back_inserter(vec));
        return vec;
      });
    }
  }
}

template <typename T>
void send(int client_fd, T message, std::function<vector<uint8_t>(T)> f) {
  //
  auto data = f(move(message));
  auto st = ::send(client_fd, &data[0], data.size(), 0);
  // HANDLE_SEND_RECV_ERRORS(st);
}
template <typename T>
T receive(int client_fd, size_t sz, std::function<T(const char *)> f) {
  //
  char buf[sz + 10];
  auto st = recv(client_fd, buf, sz, 0);
  // HANDLE_SEND_RECV_ERRORS(st);
  buf[st] = 0;
  return f(buf);
}
static std::pair<std::string, int> get_ip_port(const sockaddr_storage *addr) {
  int port;
  char buf[1000];
  if (addr->ss_family == AF_INET) {
    port = ((struct sockaddr_in *)addr)->sin_port;
    inet_ntop(addr->ss_family, &((struct sockaddr_in *)addr)->sin_addr, buf,
              sizeof(sockaddr_storage));
  } else {
    port = ((struct sockaddr_in6 *)addr)->sin6_port;
```

```
      inet_ntop(addr->ss_family, &((struct sockaddr_in6 *)addr)->sin6_addr, buf,
                sizeof(sockaddr_storage));
    }
    return {std::string(buf), port};
  }

  static struct addrinfo *addr_setup(const char *address, const char *port_no) {
    struct addrinfo hints, *result;

    memset(&hints, 0, sizeof(hints));
    hints.ai_addr = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    int status = getaddrinfo(address, port_no, &hints, &result);
    if (status != 0)
      die_with_error("getaddrinfo");
    return result;
  }

  static int setup_client_socket(struct addrinfo *result) {
    int yes = 1;
    int sock_fd =
        socket(result->ai_family, result->ai_socktype, result->ai_protocol);
    if (sock_fd < 0)
      die_with_error("socket()");
    if (setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT | SO_LINGER,
                   &yes, sizeof(yes)) < 0)
      die_with_error("setsockopt");
    if (connect(sock_fd, result->ai_addr, result->ai_addrlen) < 0)
      die_with_error("bind()");
    return sock_fd;
  }
};

int main() {
  string port;
  cout << "enter port: ";
  cin >> port;
  Client cl(NULL, port.c_str());
  cl.handle_connection();
  //
}
```

Output



```
LabNetworks/Endsem/Q1 on cur_sem [!?]
λ ./server
enter port: 3000
Here
Connected to : 127.0.0.1/3209
Connected to : 127.0.0.1/3721
```

```
LabNetworks/Endsem/Q1 on cur_sem [!?] took 15s
λ make client
g++ -o client client.cpp -lpthread
LabNetworks/Endsem/Q1 on cur_sem [!?]
λ ./client
enter port: 3000
Enter (exit/get_online/send): get_online
hi
127.0.0.1/3209


exit
```

```
LabNetworks/Endsem/Q1 on cur_sem [!?]
λ ./client
enter port: 3000
Enter (exit/get_online/send): send
Enter Message to send:
Enter Destination address: hi
```

Question 2

Code

- I have assumed they go down at 45
- This simulates the protocal based on command line argument by setting the proto value. It can be LS(OSPF) or DV(RIP)

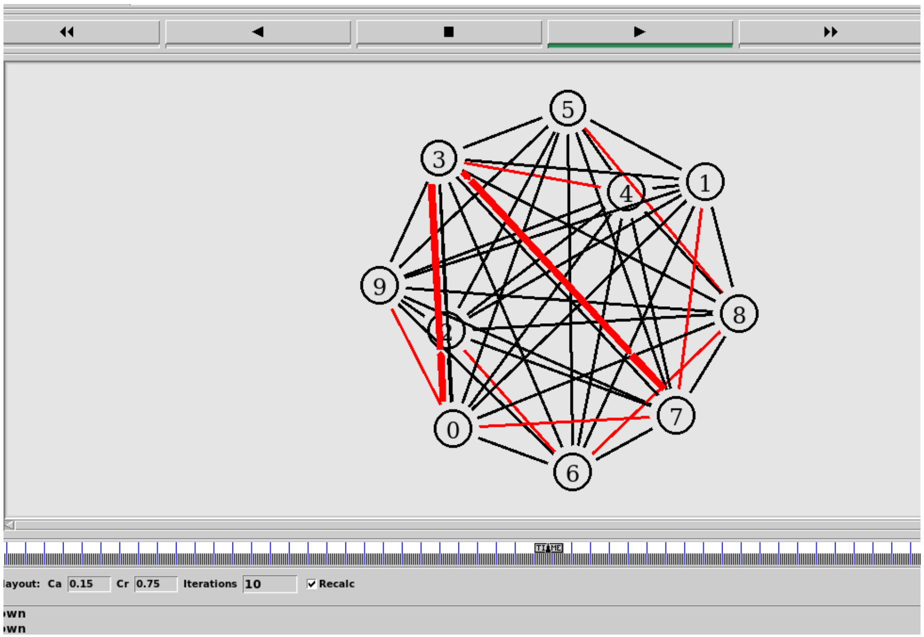
```
set ns [new Simulator]


$ns color 1 Blue
$ns color 2 Red


#Open the nam trace file
set nf [open out.nam w]
$ns namtrace-all $nf

set all_trace [open all.tr w]
$ns trace-all $all_trace

#Define a 'finish' procedure
proc finish {} {
        global ns nf all_trace
        $ns flush-trace
    #Close the trace file
        close $nf
    close $all_trace
    #Execute nam on the trace file
```

```
        exec nam out.nam &
        exit 0
}

# make a tcp connection between src and sink
proc makeTcp { src sink } {
    global ns
    set tcp [new Agent/TCP]
    $ns attach-agent $src $tcp
    set sinkAgent [new Agent/TCPSink]
    $ns attach-agent $sink $sinkAgent
    $ns connect $tcp $sinkAgent
    $tcp set fid_ 1

    set ftp [new Application/FTP]
    $ftp attach-agent $tcp
    $ftp set type_ FTP
    return $ftp
}

# make a udp connection between src and sink
proc makeUdp { src sink } {
    global ns
    set udp [new Agent/UDP]
    $ns attach-agent $src $udp
    set null [new Agent/Null]
    $ns attach-agent $sink $null
    $ns connect $udp $null
    $udp set fid_ 2

    set cbr [new Application/Traffic/CBR]
    $cbr attach-agent $udp
    $cbr set type_ CBR
    $cbr set packet_size_ 1000
    $cbr set rate_ 1mb
    $cbr set random_ false
    return $cbr
}

proc makeNodes { num } {
    global ns

    set nodes [list]
    for {set i 0} {$i < $num} {incr i} {
        set node [$ns node]
        lappend nodes $node
    }
    return $nodes
}


# Get the routing protocol to be used as command line arg
set proto [ lindex $argv 0 ]
# set protocol
$ns rtproto $proto

set N 10
# make n nodes
for {set i 1} {$i <= $N} {incr i} {
    set n($i) [$ns node]
}



# Make Mesh
# connect each node with all other nodes
for {set i 1} {$i <= $N} {incr i} {
    for {set j [expr {$i + 1}]} {$j <= $N} {incr j} {
            $ns duplex-link $n($i) $n($j) 1Mb 10ms DropTail
    }
}

set ftp [makeTcp $n(1) $n(10)]
set cbr [makeUdp $n(1) $n(8)]


$ns rtmodel-at 45 down $n(2) $n(8)
$ns rtmodel-at 45 down $n(1) $n(10)
$ns rtmodel-at 45 down $n(4) $n(5)
$ns rtmodel-at 45 down $n(3) $n(7)
$ns rtmodel-at 45 down $n(6) $n(9)
$ns rtmodel-at 45 down $n(7) $n(9)
$ns rtmodel-at 45 down $n(1) $n(8)

$ns rtmodel-at 60 up $n(3) $n(7)
$ns rtmodel-at 60 up $n(1) $n(10)


$ns at 1.0 "$cbr start"
$ns at 30.0 "$ftp start"
$ns at 99.0 "$ftp stop"
$ns at 99.0 "$cbr stop"
```

```
# call finish after 200s
$ns at 100.0 "finish"

# run simulation
$ns run
```

Output

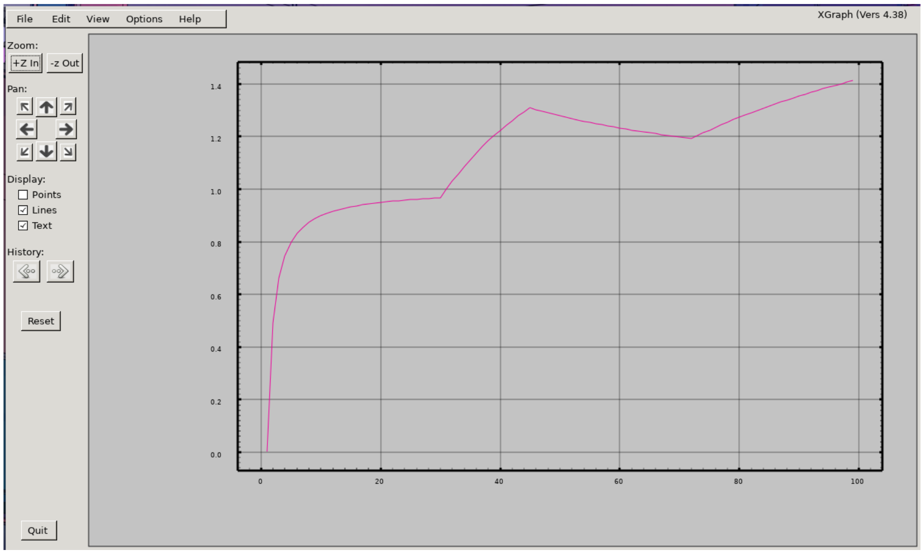
RIP



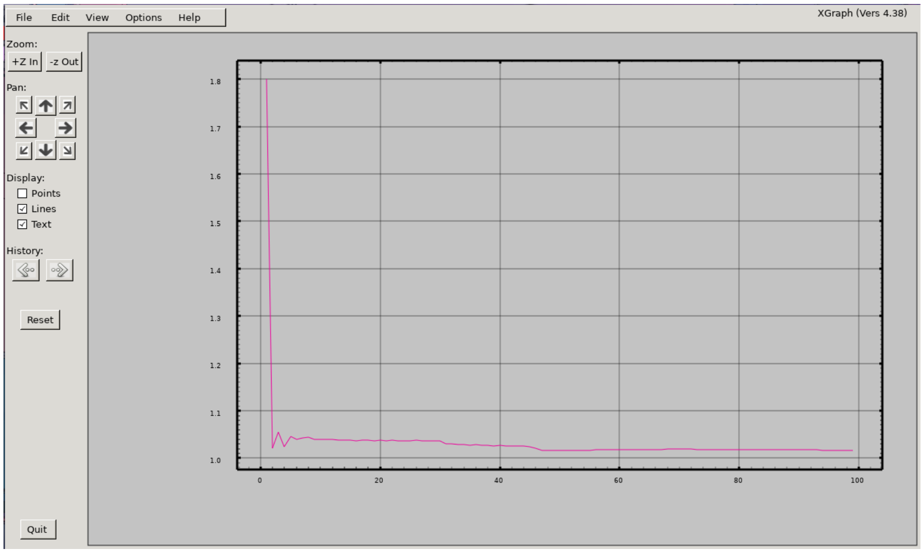
layout: Ca 0.15 Cr 0.75 Iterations 10 ☑ Recalc

wn
wn

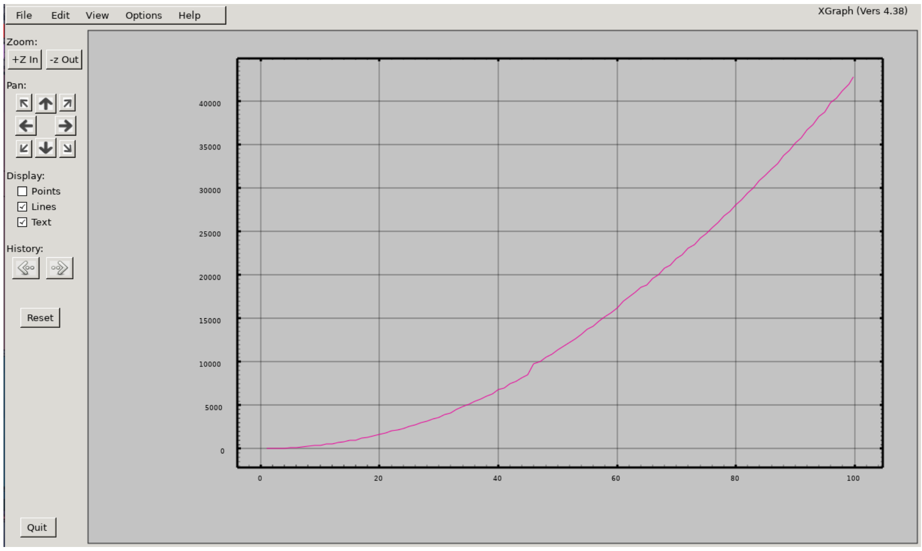
- Throughput



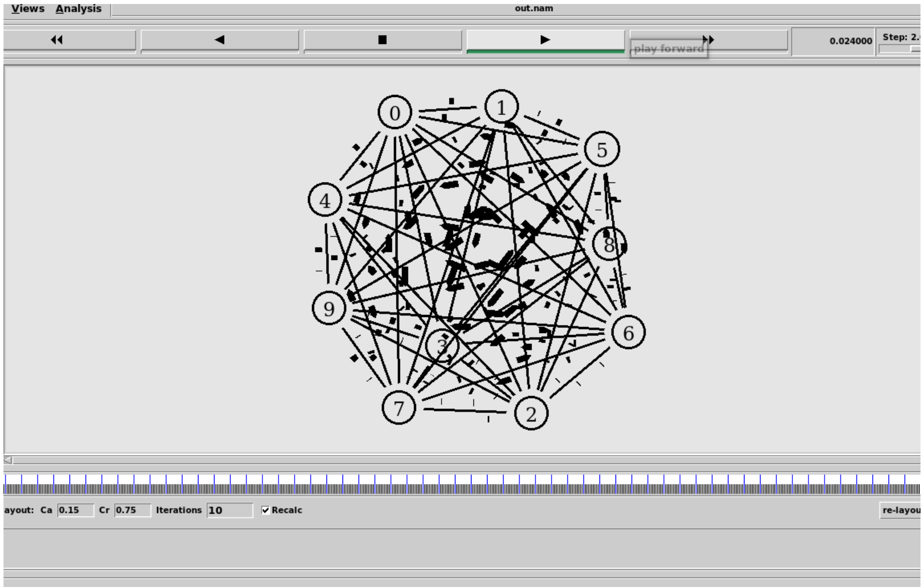
- Packet Delivery Ratio



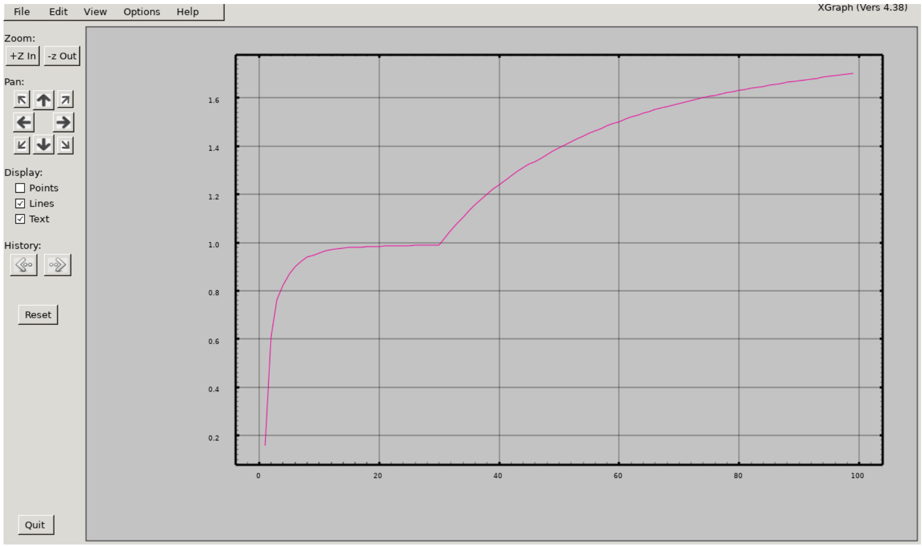
- Average delivery delay(E2E delay)



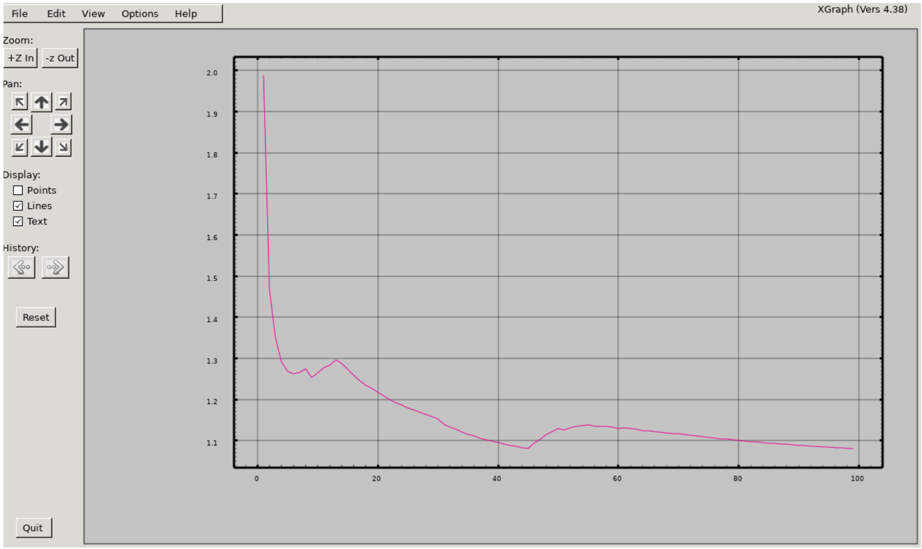
OSPF



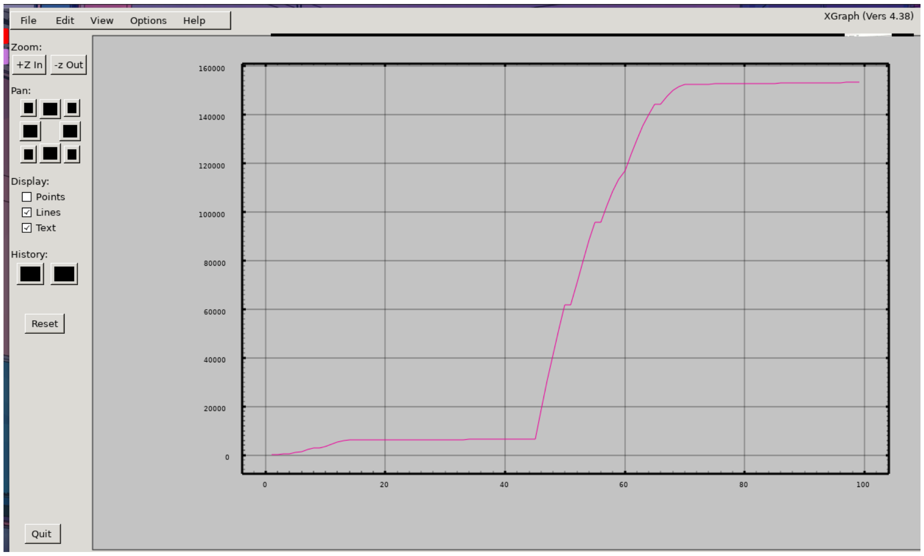
- Throughput



- Packet Delivery Ratio



- Average delivery delay(E2E delay)



Display:
☐ Points
☑ Lines
☑ Text