15 The PC LAN as a SimJava2 Model

15.1 Introduction

In this note we consider again the PC LAN first presented in lecture note 4 as a Markov process, and represented both as a GSPN and a PEPA model in subsequent lecture notes. Using simulation we are able to make the model somewhat more realistic in several respects:

- inter-event times within the model are no longer restricted to be exponentially distributed;
- since the size of the state space does not concern us quite so much we can allow more realistic buffer sizes;
- we can study the transient behaviour of the model (from "cold start" when the system is empty) just as easily as steady state measures;
- moreover, in steady state we are not restricted to considering the means but can also find, for example, the maximum and minimum value of a random variable.

Just as when we were deriving the Markovian model, we might not necessarily arrive at the correct characterisation of the system on our first attempt. However, it is often harder to establish confidence in a simulation model than a Markov process model, requiring the use of specific validation and verification techniques as described in lecture note 14.

15.1.1 Description of the system

Initially, as in lecture note 4, we aim to determine the mean waiting time for data packets at a PC connected to a local area network, operating as a token ring. We relax some of the assumptions that we made in the earlier model, but retain the restriction that a PC can transmit only when in possession of the token, and that the token circulates in *round robin* order. We now assume that whilst the inter-arrival time of packets is still exponentially distributed, packet transmission time is normally distributed and token circulation time is deterministic. In the model presented here we still assume that the PCs connected to the LAN all have the same characteristics. However, if you study the model you will see that it would be straightforward to parameterise the entity definitions in such a way as to differentiate between them.

Without the state space restrictions imposed by numerical solution of the Markov process we no longer impose that at most one packet is waiting to be transmitted from each PC. As previously though, we assume that packets that arrive when the token is already at the node must wait for the next visit of the token. Various policies are used in practice. Our assumption represents what is termed *gated service*: this means that when the token arrives at the node a notional gate is closed. The token will remain at the node long enough to transmit all those packets in front of the gate (i.e. those packets that were waiting when the token arrived) but it will leave those behind the gate (i.e. those packets that arrived while the token was at the node) until the next visit.

15.2 Modelling the system

The first task is to identify the entities to be represented in the SimJava2 model, and then the events which constitute interactions between them. An initial consideration of the system description would perhaps identify the following potential entities:

- PCs,
- Buffers,
- LAN,
- Token.

In this model some of these entities are represented implicitly. We distinguish between the PC, whose role is to generate packets, but which is otherwise independent of the network, and the "Node" which holds waiting packets, carries out the transmission and passes on/receives the token. In the Markov process we did not need to distinguish these entities but now it is necessary for two reasons. Firstly, we need to record the number of packets waiting at a node: using two distinct entities we can use the deferred queue of the node as an implicit representation of the buffer. Secondly, we wish to model concurrent packet generation and transmission. When Markovian assumptions are made we could use an interleaving approach due to the memoryless property of the exponential distribution. Since each entity has its own (single) thread of control, concurrent activity must generally be represented in distinct entities[1].

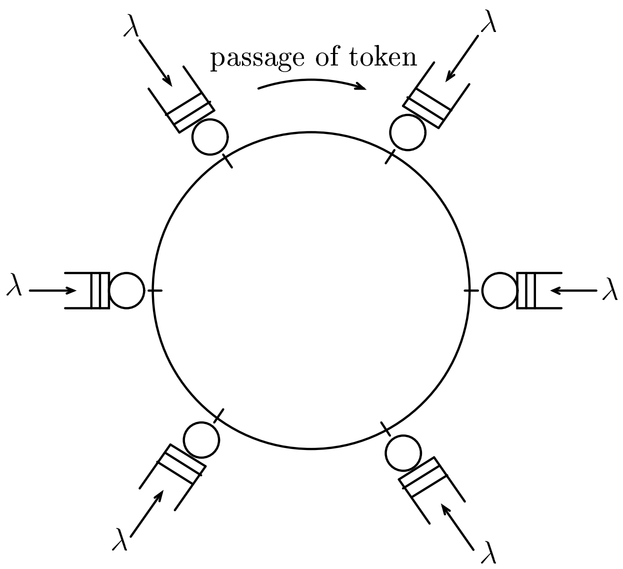

Figure 32: A six-node token ring network

The LAN is modelled only implicitly as the collection of nodes and the links between them. Once we have defined the PCs and the Nodes there is no behaviour left to be represented which indicates that having an entity to represent the LAN itself is unnecessary.

[1]The exception is that scheduling an event with a delayed time will allow the entity to concurrently proceed with its next activity.

In general, we aim to keep the number of entities within a model to a minimum. This aids both the understandability of our models and the execution times: there is significant overhead in switching between threads.

The token is modelled as an event or message rather than as an entity. In the GSPN and PEPA models of the LAN, the token was represented more explicitly and participated in a synchronised transition/activity. In SimJava (and discrete event simulation packages generally) it is difficult to represent shared activity so directly. If two entities undertake joint work, in the model it is represented (as a delay) in only one of the entities, the other being passivated until the work is complete. In SimJava this would be represented by the passive entity first sending a message to say it was ready to proceed with the shared work and then waiting for a message from the active partner, indicating that the work had been completed. Other languages, such as DEMOS, distinguish this form of interaction, by including control structures in the language which allow one entity to temporarily be treated as a resource by another entity.

Thus the SimJava entities which we use to represent the system are just `PC` and `Node`. The `Sim_entity` `PC` is shown in Figure 33, and described in detail below.

```
class PC extends Sim_entity {
  private Sim_port enqueue;
  private Sim_negexp_obj pc_hold;
  private double iatime;

  public PC(String name, int x, int y, double iatime) {
    super(name, "pc", x, y);
    this.iatime = iatime;
     enqueue = new Sim_port("enqueue", "port", Anim_port.BOTTOM, 15);
    add_port(enqueue);
    pc_hold = new Sim_negexp_obj("iatime", iatime);
    add_generator(pc_hold);
  }

  public void body() {
    int i;

    while (Sim_system.running();) {
      sim_schedule(enqueue, 0.0, 0);
      sim_trace(1,"S enqueue");
      sim_pause(pc_hold.sample());
    }
  }
}
```

Figure 33: SimJava2 entity representing the PC (generator of packets)

The entity has several parameters: in addition to the name and location coordinates, there is the mean interarrival time for the data packets `iatime`, which is used to instantiate the exponentially distributed random variable used to generate the delays between packet arrivals. Note that in general the user of a model is not necessarily the modeller and so it can be a good idea to allow the user to modify parameters of the model without editing the SimJava2 code.. The `body()` of the entity is simple—until the end of the run data packets are generated and sent as events from the port `enqueue` (to the corresponding node) and the entity waits for a sample of the random variable representing the interarrival time.

```
class Node extends Sim_entity {
  private Sim_port buffer, token_in, token_out;
  private Sim_normal_obj trans_hold;
  private int num_packets;
  private double transmean, transvar, walk;
  private boolean lr;

  public Node(String name, int x, int y, double transmean, double transvar,
                                                  double walk, boolean lr) {
    super(name, "node", x, y);
    this.transmean = transmean;    this.transvar = transvar;
    this.walk = walk;              this.lr = lr;

    buffer = new Sim_port("buffer", "port", Anim_port.TOP, 10);
    add_port(buffer);
    if (lr) {
      token_in = new Sim_port("token_in", "port", Anim_port.LEFT, 10);
      add_port(token_in);
      token_out = new Sim_port("token_out", "port", Anim_port.RIGHT, 10);
      add_port(token_out);
    } else {
      token_in = new Sim_port("token_in", "port", Anim_port.RIGHT, 10);
      add_port(token_in);
      token_out = new Sim_port("token_out", "port", Anim_port.LEFT, 10);
      add_port(token_out);
    }
    add_param(new Anim_param("num_packets", Anim_param.VALUE, "0", 1,10));
    trans_hold = new Sim_normal_obj("trans_hold",transmean,transvar);
    add_generator(trans_hold);
    stat = new Sim_stat();
    stat.add_measure(Sim_stat.WAITING_TIME);  set_stat(stat);
  }
```

Figure 34: Constructor for the `Node` entity

The entity representing the `Node` is represented in Figures 34 and 35.

Again parameters of the random variable are passed as parameters to the entity's constructor. The parameter `lr` is used to determine whether the node appears in the animation in *left-right* orientation: accepting the token on the left and passing it on to the right. Thus the constructor uses this parameter to determine just where the ports are added to the icon representing the node. Of course such visual niceties have nothing to do with the simulation aspects of the code.

```
  public void body() {
    Sim_type_p token = new Sim_type_p(2);
    while (Sim_system.running()) {
      Sim_event control = new Sim_event();
      sim_get_next(token, control);
      if (sim_waiting() > 0) {
        num_packets = sim_waiting();
        sim_trace(1,"P "+num_packets);
        for (int i = num_packets-1; i >= 0; i--) {
          Sim_event ev = new Sim_event();
          sim_get_next(ev);
          sim_process(trans_hold.sample());
          sim_completed(ev);
          sim_trace(1, "P "+i);
        }
        num_packets = 0;
      }
      sim_schedule(token_out,walk,2);
      sim_trace(1,"S token_out");
    }
  }
}
```

Figure 35: The `body()` of the `Node` entity

This model has been set up just to record the average waiting time for packets. As we can see in the `body()` in Figure 35 a default measure can be used for this and no explicit updating is necessary.

Since the `Node` maintains a single deferred queue for events, the token events (tag 2) must be selected from amongst the events representing incoming data packets (tag 0). This is done using a predicate. SimJava provides various predicate constructors. Here we use `Sim_type_p(int tag)`, which constructs a predicate to test whether an event has the specified tag value (in our case 2). This predicate can then be passed as an additional argument to several of the event handling methods. In our case we use the method `sim_get_next()` with this predicate. This has the effect of selecting an event from the deferred queue if it can find one with the matching tag, or waiting for an event with the matching tag to arrive. Events in the deferred queue with other tag values are ignored. In `Node` this means that the packets are allowed to accumulate in the deferred queue until the token next arrives.

When the token does arrive the length of the deferred queue is checked (`sim_waiting()`)—this is the number of packets currently in the buffer, the number of packets to be transmitted on this visit of the token. If this number is zero the token is immediately passed on to the next node. In fact the transmission time of the token (`walk`) is represented as a delay in the scheduled event:

```
sim_schedule(token_out, walk, 2);
```

If the value is non-zero a `for` loop is then used to send each of the packets. The animation parameter `num_packets` records how many packets are remaining to be transmitted on this visit of the token. When all packets have been transmitted the token is passed on to the next node through the port `token_out` as described above.

At the end of the simulation a special entity, the `Terminator` (see Figure 36), is used to gather the `stat` objects from each of the nodes and calculate an aggregate average waiting time. To keep it passive through the simulation run it is set to wait for a message although none will be sent. The Terminator is a modelling entity and does not represent anything in the system so it is not desirable to have it visible in the animation. Thus the animation form of the `Sim_entity` constructor is not used. On construction the `Terminator` is

```
class Terminator extends Sim_entity {
  private int LANsize;

  public Terminator(String name, int LANsize) {
    super(name);
    this.LANsize = LANsize;
  }

  public void body() {
    Sim_event next = new Sim_event();
    while (Sim_system.running()) { // kept passive by waiting
      sim_wait(next);              // for an event until the
    }                              // end of the simulation run
    double sum = 0.0;
    for (int i=1; i <= LANsize; i++) {
      sum +=
    ((Sim_system.get_entity("Node"+i)).get_stat()).average(Sim_stat.WAITING_TIME);
    }
    PC_LAN.WTavg.setText("Average Waiting Time: "+(sum/LANsize));
  }
}
```

Figure 36: The `Terminator` entity

passed a parameter `LANsize` so it knows how many nodes to collect statistics from.

The only thing remaining to complete the model is the token. The nodes have been defined in such a way as to receive the event representing the token, transmit the waiting packets (if any) and then pass on the token through the `token_out` port. However to start the simulation a token must be placed somewhere within the LAN. To achieve this we introduce an additional modelling entity, the `Starter` (Figure 37). This entity has the minimal constructor, simply calling the `Sim_entity` constructor. Its `body()` method schedules only one event, sending the token to the first node in the LAN. Since this is a modelling entity we do not wish it to have a port visibly linked to the node's icon. SimJava provides an alternative form of `sim_schedule()` to handle such cases. Instead of scheduling an event on a port, one entity can send an event directly to another if it knows the `id` of the other entity. Each entity is assigned an id when it is instantiated in the body of the simulation program (`anim_layout` in this case). The method

Sim_system.get_entity_id() returns the integer id of an entity given the string used as the name of the entity in its constructor. This is used in Starter to find the id of Node1. The starter then sends an event representing the token to that node. The entity would then be eligible for garbage collection.

```
class Starter extends Sim_entity {
  public Starter(String name) {
    super(name);
  }
  public void body() {
    sim_schedule(Sim_system.get_entity_id("Node1"), 0.0, 2);
  }
}
```

Figure 37: The Starter entity

The method anim_init() sets up the user interface to the simulation with appropriate fields and choices for user-defined parameter values. In the method anim_layout() values are extracted from the user interface and used to parameterise the constructors of the various entities. The lr parameter of the Nodes is used to arrange the nodes as a circuit for animation purposes. Note that the PCs are linked to the Nodes and the Nodes to each other but the modelling entities Starter and Terminator are not linked.

Finally, in order to set the transient and termination conditions of the run the Anim_applet method sim_setup method is overriden. This is how simulation conditions, an output analysis method and other simulation parameters are set in an animated model.

```
  // Setup the simulation's parameters
  public void sim_setup() {
    Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, warmuptime);
    Sim_system.set_termination_condition(Sim_system.TIME_ELAPSED, runtime, true);
  }
```

You should experiment with running the model under a variety of parameters. It is available in the gzipped tar file on the module web page as file PC_LAN.java, and can be viewed using the appletviewer and file PC_LAN.html.

15.3 SimJava Graph Viewer

As explained in the previous lecture note, whilst having an animation of a simulation model is a good way of gaining confidence in the model, it makes running the simulation relatively slow. Particularly when a model contains a large number of entities, this becomes an important consideration: remember that long runs and/or many replications may be necessary in order to get measures with any accuracy. Therefore it is common practice to develop a model as an animation initially and then, once it has been verified, reduce it to an application, stripping away the animation features. One of the other advantages of this approach is that report files and graph outputs can be produced from the model, which are not possible when the model is running in an applet.

SimJava2 provides support for generating graphs from any of the measures which have been collected in non-efficient mode during the simulation run. This support takes the form of a separate tool, the SimJava Graph Viewer, or SJGV, and it can greatly aid the modeller in interpreting the output from a model.

In the model `Queue_Other.java` the queue model is modified to include graph data generation. The entities are unchanged (cf `Queue_Stats.java` discussed in Lecture note 12) but the model portion of the program is as follows:

```
class Queue_Other {
  public static void main(String args[]) {
    Sim_system.initialise();
    Sim_system.set_seed_sequence(100000, 7311);
    Source customers = new Source("Customers");
    Server service = new Server("Service");
    Sim_system.link_ports("Customers", "enqueue", "Service", "arrival");
    Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, 50000);
    Sim_system.set_termination_condition(Sim_system.EVENTS_COMPLETED,
                                              "Service", 0, 500, false);
    Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 5, 0.95);
    Sim_system.set_report_detail(true, false);
    Sim_system.generate_graphs("test");
    Sim_system.run();
  }
}
```

Once the model has completed the five replications, it will generate the graph data (a serialised object) in the file `test.sjg`. This can then be viewed and manipulated by the modeller using the command

```
java -classpath /usr/java/lib/simjava/classes/SGJV SJGV test.sjg
```

This will start the tool with the graph data loaded. (Note that for long runs with several measures this may take some time). The tool has three panels.

- The top panel can be used to display summary information about the run(s) used to generate the data and summary information about the measurements calculated. Menus at the right of the panel allow the user to select the entity and measure of interest.

- The middle panel displays the graph for the chosen entity and measure. Note that the points on the graph are not the value of an individual observation made at the corresponding simulation time but the value that would be calculated for the measure had the run terminated at the corresponding simulation time.

- The lower panel offers the modeller further control over the displayed graph data. There is a button which allows the modeller to select to show the transient period for the data (based on the min-max truncation method), or the steady state. If replications have been used there is an option to choose to view one replication at a time or to have all superposed on a single graph.

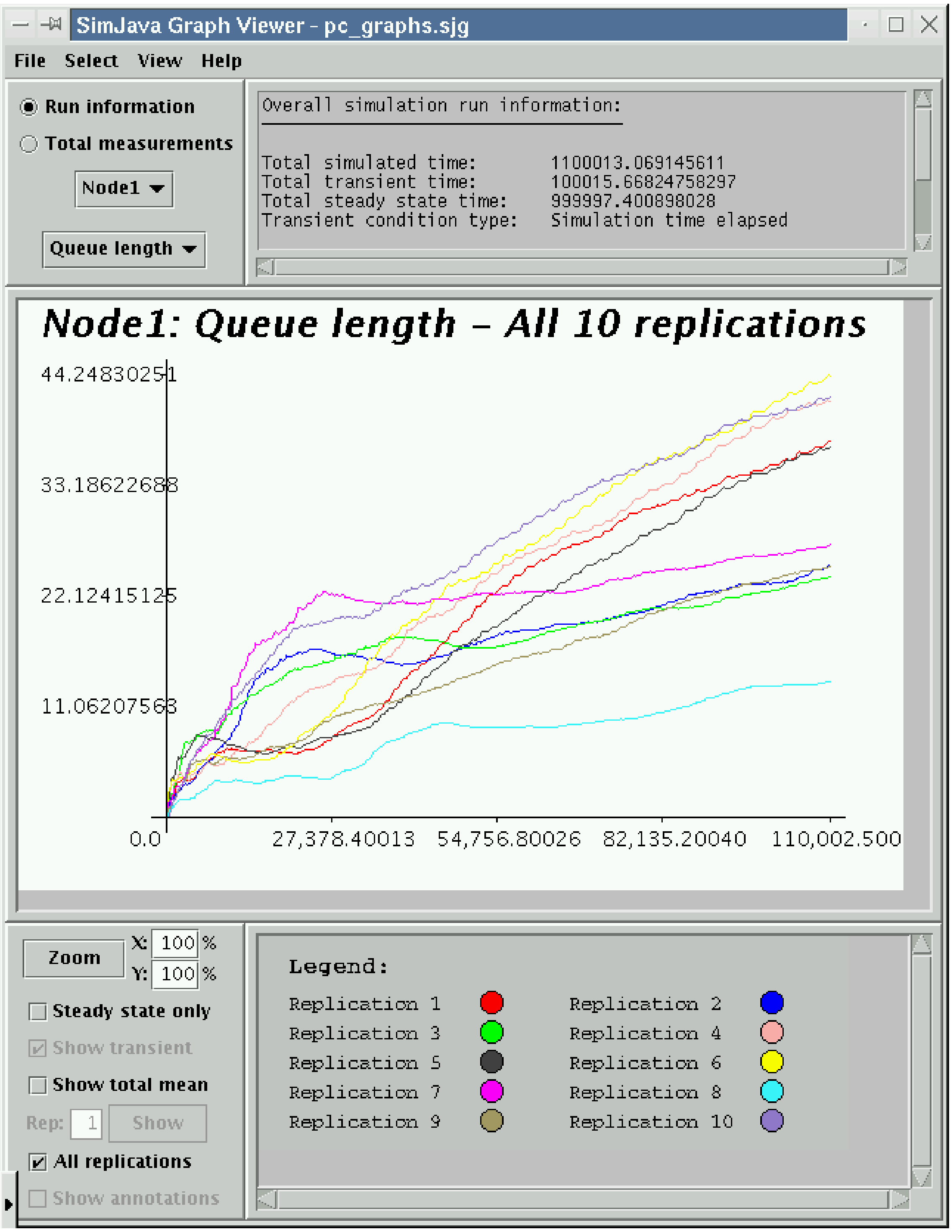

Figure 38: SJGV displaying results from the PC LAN model run as an application

Figure 38 show a screenshot of the tool displaying output from the PC LAN model run as an application. In this case the graph is displaying the queue length (average number of data packets waiting) at node 1 of the LAN. The model has been run for 10 replications and the results from each replication are shown as a separate line. The tool also offers the possibility to save a graph as in a gif file suitable for inclusion in documents (see Figure 39) .

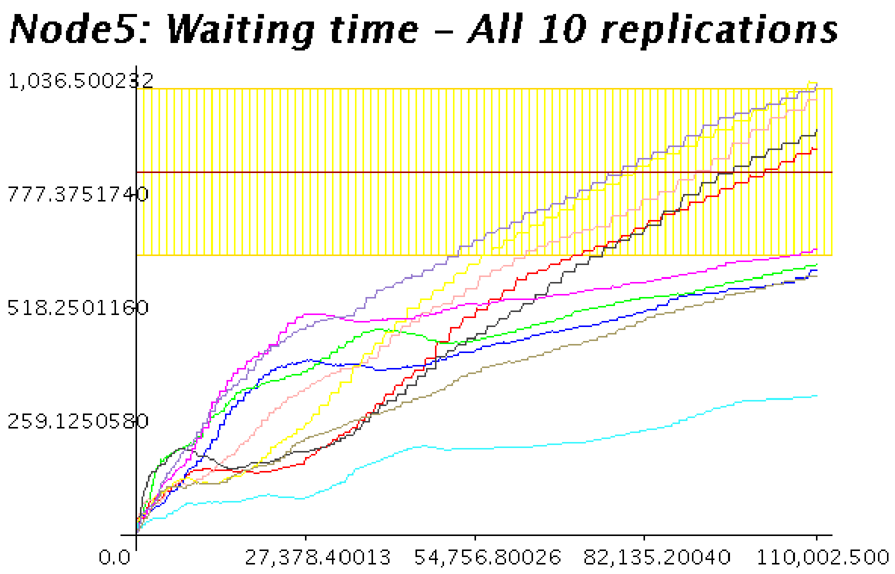

Figure 39: Graph saved from SJGV showing waiting time in PC LAN model

The PC LAN model as an application is included below. Note that the `Terminator` entity is no longer needed as we no longer need to make summary statistics available in the applet.

```
/*
 * The simple PC LAN
 */
import eduni.simjava.*;
import eduni.simjava.distributions.*;

class Pc extends Sim_entity {
    private Sim_port enqueue, test;
    private Sim_negexp_obj pc_hold;
    private double iatime;

    public Pc(String name, double iatime) {
        super(name);
        this.iatime = iatime;
        enqueue = new Sim_port("enqueue");
        add_port(enqueue);
        pc_hold = new Sim_negexp_obj("iatime", iatime);
```

```
        add_generator(pc_hold);
    }
    public void body() {
        while (Sim_system.running()) {
            sim_schedule(enqueue, 0.0, 0);
            sim_pause(pc_hold.sample());
}   }  }

class NODE extends Sim_entity {
    private Sim_port buffer, token_in, token_out;
    private Sim_normal_obj trans_hold;
    private Sim_stat stat;
    private int num_packets;
    private double transmean, transvar, walk;

    public NODE(String name, double transmean, double transvar, double walk) {
        super(name);
        this.transmean = transmean;
        this.transvar = transvar;
        this.walk = walk;

        buffer = new Sim_port("buffer");
        add_port(buffer);
        token_in = new Sim_port("token_in");
        add_port(token_in);
        token_out = new Sim_port("token_out");
        add_port(token_out);
        trans_hold = new Sim_normal_obj("trans_hold", transmean, transvar);
        add_generator(trans_hold);
        stat = new Sim_stat();      stat.add_measure(Sim_stat.WAITING_TIME);
        stat.add_measure(Sim_stat.QUEUE_LENGTH);     set_stat(stat);
    }
    public void body() {
        Sim_type_p token = new Sim_type_p(2);

        while (Sim_system.running()) {
            Sim_event control = new Sim_event();
            sim_get_next(token, control);
            if (sim_waiting() > 0) {
                num_packets = sim_waiting();
                for (int i = num_packets-1; i >= 0; i--) {
                    Sim_event ev = new Sim_event();
                    sim_get_next(ev);
                    sim_process(trans_hold.sample());
                    sim_completed(ev);
                }
```

```
                num_packets = 0;
            }
            sim_schedule(token_out,walk,2);
}    }    }

class Starter2 extends Sim_entity {
    public Starter2(String name) {
        super(name);
    }
    public void body() {
        sim_schedule(Sim_system.get_entity_id("Node1"), 0.0, 2);
    }
}
public class PC_LAN2 {
  public static void main(String args[]) {
    int num_pcs=6, i;
    double iatime=25.0, transmean=4.2, transvar=0.5, walk=1.0;
    int warmuptime=10000, runtime=110000;

    Sim_system.initialise();
    // Add entities
    Starter2 starter = new Starter2("Starter");
    for (i=1; i <= num_pcs; i++) {
        Pc pc = new Pc("PC"+i, iatime);
        NODE node = new NODE("Node"+i, transmean, transvar, walk);
    }
    // Link entities
    for (i=1; i <= num_pcs; i++) {
        Sim_system.link_ports("PC"+i, "enqueue", "Node"+i, "buffer");
        if (i < num_pcs)
            Sim_system.link_ports("Node"+i, "token_out", "Node"+(i+1), "token_in");
        else
            Sim_system.link_ports("Node"+i, "token_out", "Node1", "token_in");
    }
    // Setup the simulation's parameters
    Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, warmuptime);
    Sim_system.set_termination_condition(Sim_system.TIME_ELAPSED, runtime, true);
    Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 10, 0.95);
    Sim_system.set_report_detail(true, false);
    Sim_system.generate_graphs("pc_graphs");
    Sim_system.run();
}    }
```