CHAPTER 5: IMPLEMENTATION AND TESTING

Chapter 5 discusses the implementation and testing phase that need to be done for the simulator. During the implementation phases, classes with important data items and its attributes will be shown. Together with that, explanation of those attributes and methods contained within the classes are done. The classes are implemented with the Java programming language codes to fulfil the simulation criteria.

In the testing phase, each class will be tested on its functionality and behaviour of its operations. Testing is important to ensure the transformation processes work well in the designed simulator.

5.1 The modules

![Diagram](image)

*Figure 5.1: Inter-communication between modules and classes*
In this simulation model, each object belongs to a specific set of files where its Java implementation is written. Each set of files is packed into a package that might contain one or more classes. Each class includes the declaration of data members and the body of the simplest function members of that set of objects.

Classes with common purposes and functionality are gathered into a “module” in general or “package” in Java. In the following section, a description will be done on each package.

5.1.1 SimulatorGlobal Module
This module contains files related to general programming definition, which includes general programming definitions for this ATM network simulator, example ATM cells, queue, object indicator, and the synchronised timing clock for the simulator.

5.1.2 Link Module
This module will implement a network link, which is used to communicate between end system and switch in ATM network. ATM cells are able to flow in the link. This module consists of two classes: link class and link end point class. In object-oriented world, both link and link end point classes are objects. The link class composed of two link end point classes.

5.1.3 Buffer Driver Module
In actual implementation, buffer is considered as part of a switch component. Therefore, in this simulation, a temporary switch that named as buffer driver is built. It drives the buffer to operate with its congestion control mechanism.

In buffer class, important function members are applied. Early packet discard policy is applied into the buffer queue to make sure the buffer capacity is not overflow before enter and leaving a switch. Since buffer is a common object that can be use as an input buffer or an output buffer. Therefore, a buffer class is reusable. The buffer would react towards congestion alert state and discarding state where the cells are forced to discard from the buffer queue. The flow of ATM cells in buffer is synchronised where cells entering and leaving the buffer could be performed at the same time.
5.1.4 Testing Module

This module is built temporarily for testing purposes where dummy ATM cells are being generated from an operation class. Besides, another test class is built for interfacing component objects of this simulator.

ATM cells are transmitted into the link and being transferred towards the switch's buffer at a specific link rate. For connection between an end system and a switch, the link speed rate is fixed at an approximate value of 100 Mbps, and the link end point for end system is fixed as zero value (0). For connection between switches, the link speed rate is fixed 155 Mbps and the link end point value is being determined between the switches. By using object-oriented approach, the constructor class of link can be overloaded for two differently used purposed, one with fixed link end point at one end and another with two unfixed link end points at the end of the link.

By using a buffer class, a switch can have as many buffer as possible, where each of the buffer has its own identity but having the same functionality.

5.2 Class Implementation

The class implementation consists of many classes in this simulator, which is grouped into packages using Java approach. They are sub-divided into a simulator global package, a link package and a switch package.

5.2.1 Package: SimulatorGlobal

This package consists of general information for the execution of a simulator, which includes ATM cell, clock, and queue.

AtmCell.java

The ATM cell class is a data resource class used for the components includes link and buffer throughout the simulator. As such, it contains attributes needed by the operation of all the executor classes.
The header for the ATM cell class is as follows:

```java
public class AtmCell {
    protected int VPI; // virtual path indicator
    protected int VCI; // virtual channel indicator
    protected int CLP; // cell loss priority
    protected int PT; // payload type, as follows:
    /*
     * Payload type set to:
     * 0 - last data cell
     * 1 - not last data cell
     * 3 - backward RM cell
     */
    // following attributes for RM cells
    protected float MCR; // minimum cell rate, set by user
    protected float ACR; // RESERVED
    protected float ER=0; // RESERVED
    protected boolean CI=false; // RESERVED
    protected boolean NI=false; // RESERVED
}
```

There are two groups of attributes. The general information for a ATM cell and another group is for information of RM cell. The general attributes of ATM cell includes the Virtual Channel Identifier (VCI) and Virtual Path Identifier (VPI) values, the Cell Loss Priority (CLP) and the Payload Type (PT). The payload type is used by the ATM end system to determine how to handle incoming cells, as described in the previous section.

The other set of attributes relate to RM cells are the values for the current Allowed Cell Rate (ACR) of the end system, the Minimum Cell Rate (MCR) of the end system, the Explicit Rate (ER) value set by the network, and the Boolean flags for Congestion Indication (CI) and No Increase (NI).

The methods provide access to the value of attributes stated above.

---

**QueueNode.java and Queue.java**

The Queue and Queue Node classes work together to provide the Queue abstract data type.

The header for QueueNode class is as follows:

```java
public class QueueNode {
    private Object data; // data
    private QueueNode next; // pointer to next object
}
```

And the header for Queue class is as follows:
public class Queue {
    private QueueNode firstQueueNode; // first node in the linked list
    private QueueNode lastQueueNode;  // last node in the linked list
    private int numberOfNodes;        // number of nodes
}

The Queue class uses the Queue Node class to make a link list of objects that is the queue. Public methods in the Queue class allow for objects to be inserted at the front or back, objects to be read from a specific position, objects to be removed from the front, back or centre, as well as the number of objects in the queue and whether the queue is empty or not.

**Indicator.java**

The Indicator class is a utility class that functions to indicate a signal of specific meaning so that other object could take note of that meaning. Example, a flag to indicate the cell has left the queue.

The header of the Indicator Class is as follows:

```java
public class Indicator {
    protected int numObjects=0;
    protected int numObjectsReady=0;
}
```

The indicator class maintains an attribute for the number of objects registered to it. It also maintains an attribute for the number of objects that have signalled to it.

The methods of this class enable registration of methods or objects, the signalling of methods or objects and the resetting of the signals for all objects.

**GlobalClock.java**

The global clock class maintains a general clock for referencing purpose for the simulator. The clock will strike after an appropriate timing based on the indication of multithreading methods.

The header of the class is as follows:

```java
public class GlobalClock {
    // time attributes
    protected long tick;
    protected static float usecsPerTick = 0.01f;
```
protected long maxTicks = Long.MAX_VALUE;

//control attributes
protected boolean clockStop = true;
protected boolean tickHappens = false;

//object attributes
protected CheckList signalledThisTick = new CheckList();

The time attributes includes the tick time, the value of a tick in microseconds. In this simulation, a default value of 0.01 microseconds is used for a tick. Control attributes are used to indicate the clock stop and when a tick strikes. The object attributes maintains check list on objects that are bounding to the clock.

Public methods included in this class are methods that enable objects to register themselves with the clock, to let the objects to indicate signal when ready for the next tick time and for getting approval from the clock to proceed to the next tick. Besides, there are also methods for querying conversion of clock tick time in microsecond and vice versa.

5.2.2 Package: Link

Link package consists only two main classes, which maintain the link object during run time.

Link.java

This class maintains a conceptual model for a link, which is a composition class created from the link end point class. The header of link class is as follows:

```java
public class Link extends Thread {
    //link attribute
    protected String name;
    protected LinkEndPoint endPoint0, endPoint1;
    protected GlobalClock clock;
    protected int objectId;
    protected int delay;  // in ticks
    protected float cellRate;  // in cells per tick

    //link end point attribute
    protected float rate;   // in Mbps
    protected float length; // in KM
}
```
The attributes included are objects of link end point and clock. Besides, it consists of link delay and cell rate value of that particular link, which are calculated from the link end point attribute of rate and length.

As the link class is an extension of the Thread class, it is itself a Thread. Therefore, it executes within a separate thread within the entire simulator. Its transactions run parallel with other Thread objects in the simulator and being executed concurrently.

**LinkEndPoint.java**

The link end point class maintains the queues process in a link. The header file is as follows:

```java
public class LinkEndPoint {
    //link end point attributes
    private Queue inputQueue = new Queue();
    private Queue outputQueue = new Queue();
    private Queue queueNumberOfCells = new Queue();
    private Indicator indicatorToGet, indicatorToInsert; //for the end to let
    //the link know that is has done its job
    private AtmCell atmcell;
    private boolean readyToGet = false; //indicate whether ready or not
    private float cellCreditRemaining = 0;
    private int intCellCreditRemaining = 0;
    private int numCellsEntered = 0;

    //link attribute
    private float cellRate;
    private int delay;
}
```

Queues and indicators are initialised within this class for the processes to invoke. ATM cells are inserted into the input queue only when the link is ready to get cells. Therefore, the indicator is needed to signal the source to pass in cells. Cells would enter the link in a sequential way based on the cell rate calculated by the link. Therefore, cells would enter the link in a manner of

```
cellCreditRemaining = cellCreditRemaining + cellRate;
intCellCreditRemaining = (int) cellCreditRemaining;
cellCreditRemaining = cellCreditRemaining - intCellCreditRemaining;
```

where only when the cell rate hits a value > 0, then a cell would be inserted into the link. This keeps on occurring when the time strikes to next tick.
Another method is used to transfer the cells from input queue to output queue of the next link end point based on the cells available in the input queue. When the output queue is not empty, the link end point would indicate to get for the destination to fetch the cells over.

### 5.2.3 Package: Testing

The classes within this package acts as the drivers to test the simulator using generated ATM cells at the source. Therefore, an operation class is up to inject cells into the link.

**Operation.java**

This operation class acts as ATM cells generator and the header file is as follow:

```java
public class Operation extends Thread {
    //attribute used
    protected AtmCell cell;
    protected Queue outgoingCellQueue;
    protected float runningCellCredit = 0f; //continuous track of cell credit
    protected int cellCredit = 0; //number of cells allowed to be released
    protected float ACR = 0f; //ACR value obtain from the cell

    //General attributes
    protected String name;
    protected GlobalClock clock;

    //Connection information attributes
    protected Link connectedLink;
    protected int linkEnd = 0;
    protected int objectld;
}
```

This class also acts as a single thread, which is acting as a temporary end system. ATM cells are initialised to make sure they have values in the cells, includes VPI, VCI, and ACR value. Besides, this class needs to initialise the link, which it is connected to.

Methods are used to send ATM cells into the link, only when there is signal from the link indicates ready to insert.

### 5.2.4 Package: Switch

The switch package consists of a buffer class and a buffer driver, which acts as a switch that drives the buffer.
**Buffer.java**
The buffer class has its method in managing its own buffer and consists of attributes useful for itself. The header file is as follows:

```java
public class Buffer {
    //connection attribute
    protected AtmCell atmCell;
    protected Queue bufferQueue; // cells contain in this buffer
    protected Link connectedLink; // a link connected to buffer
    protected int linkEndPoint; // link end point 0 or 1

    //buffer attributes
    protected int max_buff_size; // maximum buffer size in cells
    protected int curr_buff_size; // buffer size occupied so far in cells
    protected int total_cells_lost; // total cells discard in this buffer in cells
    protected int EPD_Threshold; // user input EPD threshold

    //control attributes
    protected float ERValue; // explicit rate value
    protected float rate; // cell rate
    protected float linkRate; // link rate
    protected int numCellsEnteredBuffer=0; // cells entered buffer
}
```

Connection attributes includes ATM cells, buffer queue and connected link with link end point, which need to be initialised before run time. Buffer attributes include buffer size, threshold value, current buffer size and total cell lost. Finally, control attributes are needed when congestion occurs and some values in the cell could be change, example is the ER value of a forward RM cell.

The methods included for the buffer class are fetching ATM cells, sending ATM cells, alerting congestion, discarding cells, checking for connection above fair allocation and checking for forward RM cells.

**BufferDriver.java**
The buffer driver class is a temporary switch that drivers the buffer to execute. In this simulation, a switch composes of many buffers. Therefore, the header of the file is as below:

```java
public class BufferDriver extends Thread {
    //Buffer attribute
    protected Buffer connectedBuffer01, connectedBuffer02;
    protected int max_buff_size; // maximum buffer size in cells
    protected int EPD_Threshold; // threshold value in cells
```
/General attributes
protected String name;
protected GlobalClock clock;
protected int objectId;

//connected link attribute
protected Link connectedLink01, connectedLink02;
protected int linkEndIn;
protected int linkEndOut;
}

Attributes used includes in the buffer driver class are connected link and buffer attributes. Besides, the class also has its general attributes as it drives itself as a single thread communicating with the clock.

The only method performed in this class is switching cells from one end of the switch to another end of the switch.

### 5.3 Testing

Testing is done step by step from classes to classes, module to module, and finally at the whole simulator itself. This object-oriented link and buffer management methodology could be tested in various conditions, such as by using different dummy ATM cells that consist various type of data. The successfulness of the testing will prove that this simulation model could simulate in a proper manner.

#### 5.3.1 Class Testing

Testing had been done of several important classes such as the queue class, link end point class, link class, buffer class and clock class. The testing processes will be discussed further in the following sections.

#### 5.3.1.1 Queue Class Testing

The queue class was tested by inserting object by object into the queue. The object could be any type, such as Integer, Boolean, Character as well as ATM cell object. From this test, an amount of objects was inserted into the queue. The result proved that the correct objects were inserted in the sequential way into the queue. Besides, the queue was tested with the removing
of objects from the front and back of the queue. Furthermore, the queue was tested on collecting the details for an object based on its position in the queue.

5.3.1.2 Link End Point Testing

The link end point has two types of queues – input queue and output queue. Since the queue has tested, therefore, ATM cell objects were used to send into the input queue of the link end point. Successfully, the cells entered the input queue sequentially. Besides, a queue of cells at the output queue was fetched from the queue in a sequential way. From this test, it proved that the link end point is working.

5.3.1.3 Link Class Testing

Link class was tested with an amount of ATM cells to ensure that that amount of ATM cells actually gone through the link end point's input queue and transferred to the other link end point's output queue. From this test, it is proved that the object-oriented concept for the link module of this ATM network simulator is successfully running.

5.3.1.4 Buffer Class Testing

The integration between the link and the buffer was tested when the test class initialised a link and a buffer is setup. An amount of ATM cells were pumped into the link. The cells were be able to pass through the link and finally fetched by the buffer after a value of ticks time. Therefore, this proved that the link class is successfully connected with the buffer class. Besides, the indicator class was proved to be function as when the output queue of the link indicated a signal “ready to get”, the buffer class able to detect the signal and started to fetch cells accordingly.

The congestion control functions and method for the buffer will be further discussed in the section 5.3.2 of Functional Testing.

5.3.1.5 Cell insertion Testing

A testing for the cell insertion into the link and is done with some test data. For example the link speed, link length, and ATM cells.
If the link is set to 100 Mbps and the link length is 0.1KM, the link cell rate is 0.23584905 cells/tick and the delay for each cell to stay in the link is 33 ticks. Therefore, from section 3.2.3 Link Traffic Flowing methods, at each tick, only 0.23584905 amount of cell would enter the link, but could not consider as a full cell that has entered the link. For the first time, only after 5 ticks, then one cell would be considered has entered the input queue of the link (Figure 5.2).

![Diagram showing cells flow in the link.](Image)

Figure 5.2: Diagram showing cells flow in the link.

After a delay of 33 ticks after the simulator begin execution, cells from the input buffer could be transferred to the output queue at the other end of the link. Therefore, the buffer fetches the cell once the link indicated that there is cell waiting in the output queue of the link. This simulation of cells would go through for a period of time until it ends.

### 5.3.2 Functional Testing - Congestion Control Mechanism

Congestion control test on the early packet discard policy is done onto the buffer while function of fetching ATM cell is executing. This could be done at both input and output buffer of a switch with certain assumption towards the simulation. ATM cells travel in and out from the buffer are synchronised using the tick time. Cell could leave or enter the buffer at the same tick time.

First assumption: Congestion could happen when the switch is switching at a lower rate compare to the link rate. Meaning that each time when the cells are fetch from the link, it would cause congestion at the input buffer of the switch as the amount of cells leaving the buffer queue is lesser than the amount of cells entered the buffer queue.
Second assumption: Congestion could happen at the output buffer if the switch's switching rate is too high compare to the link connected at the output buffer. Also, when there are too many applications send data to the same target destination. Meaning that, the output port at the switch is highly demanded and needs to transfer out the cells in a high rate. Unfortunately it is difficult to send out the cells to the link, if the link rate is slower.

Testing on the second assumption is much more logic as it usually happens in switches. But there is another assumption for application flows in the link as application is identified based on its virtual channel identifier (VCI) and virtual path identifier (VPI). Even though there are various application data pumped into the input buffer, there is possibility that congestion would not occur at output buffer as the cells are already destined to the addressed port of the switch.

Therefore, congestion could occur at either input or output buffer for a switch. Testing has been done at both the output and input buffers. The result indicated that, with the function of fetched ATM cell, it would perform congestion control checking whenever a cell is fetched into the buffer queue. Therefore, both input and output buffers are the same. The object-oriented approach enables the reuse of buffer class. A switch could initialise as many buffers as possible.

So, test cells would have at least two types of VPI/VCI to identify there are two applications sending data along the link towards the input buffer. The operation class would first generate the cells consequence sending two cells of that application, then, another cell with payload type 0 would be generated to identify the last data cell for that segment (in this simulation, 3 cells in a segment is used instead of the original TCP segment size with 32 ATM cells in one segment). After generating every 3 cells for that application, it would generate a forward RM cell, which is identified by the payload type of 2.

The buffer size in this simulation is set to 15 (instead of 500 to 1000 cells in actual environment), and EPD threshold is set to 9 (an amount lesser than the buffer size). Cells are kept on pumping into the buffer and had made the buffer capacity is higher than the buffer threshold when the global clock hits up to the point of 150 ticks.
At this point, forward RM cells from the application that is above their fair share in the buffer is alerted and its explicit rate (ER) value is changed to a lower rate, which is as low as 0.0296875 cells/tick. At the same time, the buffer starts to prevent new cells of a new segment from entering. If the incoming cell is not a last cell and it belongs to the application that using bandwidth that is above the fair share, all the cells from the back of the buffer queue until the application's first last data cell would be discarded (except the data cell).

After the first discarding attempt, the next in-coming cell would have re-checking on the buffer capacity, if it is still above threshold value, the alert on forward RM cell would not only target on application that using bandwidth above their fair share, but each and every forward RM cells appear in the buffer. The end-systems should decrease their cells transmitting rate into the buffer. From these test data, the cells entering into the buffer is done in sequential way as shown in Figure 5.3:

In Figure 5.3, since the next coming cell is not a last data cell and it is an application with VPI/VCI 1 above the fair share, then the forward RM cells is alert by changing its ER value. At the same time, the 10th and 9th cell are being discarded.

These test could prove that the modelled ATM network simulator is functioning properly with the used of multithreaded simulator global clock and functions are well defined in each classes of objects.
Figure 5.3: Cells flow into buffer until capacity is above threshold

5.4 Chapter Summary

This chapter discusses the two phase of development process – implementation and testing.

In the implementation phase, the conceptual and abstract designs were transformed into programming language codes. The simulator components are carefully designed into classes. The codes are compiled into executable programs with the revenant programming language compiler.
Testing is done at the final phase of development. The components are tested for their functionality and behaviour during their operation. The test has proven that the simulator is successfully executed with the designed model.

In the concluding chapter, the final results of the project will be discussed.