GROUP 16

Arpan Konar - 160101014

Arpit Gupta - 160101015

Assignment 2

OVERVIEW

The program will consist of three phases:

- Phase 1: Traffic Generation.
- Phase 2: Packet Scheduling.
- Phase 3: Packet Transmission.

All three phases will take place at the beginning of each time slot. Initially, all the packet queues are assumed to be empty.

HOW TO RUN

Compilation

Run the following command in the terminal to compile all the java files in the folder:

```
$ javac *.java
```

Execution

Run the following command in the terminal to run the program with necessary arguments:

```
$ java PacketSwitchController switchportcount buffersize packetgenprob queueType knockout outputfile maxtimeslots
```

Example:

```
$ java PacketSwitchController 8 4 0.5 INQ 4 output.txt 10000
```

Note: It will return an error if incorrect inputs are given, for example, if **switchportcount** is given as double instead of integer or **packetgenprobability** is more than 1 or less than 0.

The output will be appended to output file.

CODE DESCRIPTION

File Structure

Constants: Contains the constants and enum.

InputPort: Java Class representing an InputPort.

OutputPort: Java Class representing an OutputPort.

Packet: Java Class representing a Packet.

PacketSwitch: Java Class representing a switch with input and output ports.

Util: Helper functions for output and random value generators.

PacketSwitchController: Main entry point of the program, initializes other classes and takes input.

Traffic Generation

```
//Phase 1: Corresponds to traffic generation.
//Generate packets for all the input ports with given probability
for(InputPort inputPort: inputPorts){
   //Buffer already full, cannot take a packet unless one is transmitted
   if(inputPort.isBufferFull()) continue;
   boolean shouldGeneratePacket = util.generatePacketWithProbability(packetGenProbability);
   if(shouldGeneratePacket){
       int outputPortIndex = util.getOutputPortIndex(portCount);
       //Create new packet
       Packet packet = new Packet(inputPort, outputPorts.get(outputPortIndex), time);
       //Allocate packet to input port's buffer
       inputPort.addToBuffer(packet);
   }
}
```

shouldGeneratePacket determines if we should generate a packet with the given probability. The corresponding function generates a random integer less than 100 and determines if it is less than prob*100.

```
//returns 1 to generate packet, otherwise 0
//Rounding probability to 2 decimal places
boolean generatePacketWithProbability(double prob){
   prob = (double)Math.round(prob * 100d) / 100d;
   int rand = random.nextInt(100);
   double val = getAverage(rand, 100);
   return val <= prob;
}
```

Packet Scheduling

This phase contains three cases according to the queueType given. The queue type can be one of three types **INQ/KOUQ/ISLIP**

INQ Scheduling

First we create a **list of packets** contending for each output port.

```
//Creating temporary list for each Output Port
List<List<Packet>> outputPortContention = new ArrayList<>();
for(int i  = 0;i<portCount;i++){
   List<Packet> tempList = new ArrayList<>();
   outputPortContention.add(tempList);
}

for(InputPort inputPort : inputPorts){
   Packet packet = inputPort.getPacketAtIndex(0);
   if(packet!=null){
       OutputPort destinationPort = packet.getDestinationPort();
       int outputPortIndex = outputPorts.indexOf(destinationPort);
       //Adding it to that list
       outputPortContention.get(outputPortIndex).add(packet);
   }
}
```

Then we **generate a random index** from the list of the packets for each output port. We remove the selected packet from the input port's buffer and add it to the output port's buffer.

```
//Choose one randomly from each output port's list to transfer
for(int i = 0;i<portCount;i++){
   if(outputPortContention.get(i).isEmpty()) continue;
   int packetIndex = util.generatePacketIndex(outputPortContention.get(i).size());
   //Get that packet
   Packet packetSelected = outputPortContention.get(i).get(packetIndex);
   //Remove from input port's buffer and add to output port's buffer
   packetSelected.getSourcePort().removeFromBuffer(packetSelected);
   packetSelected.getDestinationPort().addToBuffer(packetSelected);
}
```

KOUQ Scheduling

We again create **a list of packets** contending for each output port.

```
//Creating temporary list for each Output Port
List<List<Packet>> outputPortContention = new ArrayList<>();
for(int i  = 0;i<portCount;i++){
   List<Packet> tempList = new ArrayList<>();
   outputPortContention.add(tempList);
}

for(InputPort inputPort : inputPorts){
```

```
    Packet packet = inputPort.getPacketAtIndex(0);
    if(packet!=null){
        OutputPort destinationPort = packet.getDestinationPort();
        int outputPortIndex = outputPorts.indexOf(destinationPort);
        //Adding it to that list
        outputPortContention.get(outputPortIndex).add(packet);
        inputPort.removeFromBuffer(packet);
    }
}
```

Then **for each output port**, we perform the following steps:

- **Sort** the packets according to their arrival times.

```
//Sort the packets according to arrival time
outputPortContention.get(i).sort(Comparator.comparingInt(Packet::getArrivalTime));
```

- **Choose K packets randomly** after that if more than K packets are contending for an output port.

```
//First min(knockout, total packets for output port) packets are to be considered
int K = knockout < outputPortContention.get(i).size()? knockout :
outputPortContention.get(i).size();
if(knockout < outputPortContention.get(i).size())
   numberOfPortsWherePacketDropped++;

//Randomly choose K indexes if more packets than K are in contention
for(int j = 0;j<K;j++){
   int randomPacketIndex = util.generatePacketIndex(outputPortContention.get(i).size());

   packetsToBeTransmitted.add(outputPortContention.get(i).get(randomPacketIndex));
   outputPortContention.get(i).remove(randomPacketIndex);
}
```

- If the number of packets selected to send to output buffer is more than the output buffer can accomodate, drop the packets which cannot be accomodated.

```
//Max packets that the output port buffer can accomodate
int maxPackets = bufferSize - outputPorts.get(i).getOutputBufferSize();

//Remove from input port's buffer and add to output port's buffer
int j = 0;
for(j = 0;j<Math.min(maxPackets, K);j++){
   Packet p = packetsToBeTransmitted.get(j);
   p.getDestinationPort().addToBuffer(p);
}
```

ISLIP Scheduling

First, we initialize **accept pointers** and **grant pointers** for round robin ISLIP scheduling. We also create temporary data structures for performing the operations.

```
//Grant pointers and accept pointers
List<Integer> grantPointers = new ArrayList<>();
List<Integer> acceptPointers = new ArrayList<>();

//Data structure to allocate a packet to each output port
List<Packet> outputPortPacketAllocated = new ArrayList<>();
for(int i = 0;i<portCount;i++){
   outputPortPacketAllocated.add(null);
   grantPointers.add(0);
   acceptPointers.add(0);
}

//Temporary Data structure to keep track of valid requests in each iteration.
List<List<Packet>> requestLists = new ArrayList<>();
for(InputPort inputPort: inputPorts){
   List<Packet> tempInputBuffer = new ArrayList<>(inputPort.getInputBuffer());
   requestLists.add(tempInputBuffer);
}
```

The three phases are performed in each iteration: Request Phase, Grant Phase and Accept Phase:

- **Grant Phase -** Grant requests to input ports

```
for(int i = 0;i<portCount;i++){
   List<Packet> inputRequests = requestLists.get(i);
   //If the index of the output port number is >= GRANT POINTER, allocate it
   for(Packet p: inputRequests){

       int outputPortIndex = outputPorts.indexOf(p.getDestinationPort());
       if(outputPortPacketAllocated.get(outputPortIndex) == null){
           outputPortPacketAllocated.set(outputPortIndex, p);
       }
       else if(i >= grantPointers.get(outputPortIndex)){
           int allocatedInputIndex =
inputPorts.indexOf(outputPortPacketAllocated.get(outputPortIndex).getSourcePort());
           if(i < allocatedInputIndex)
               outputPortPacketAllocated.set(outputPortIndex, p);
       }
   }
}
```

- **Accept Phase -** Input ports accept one among many accepted requests.

```
for(int i = 0;i<portCount;i++){
   //Get the inputPortIndex
   //For each input port allocate the first pkt with output port index >= accept pointer
```

```
    Packet p = outputPortPacketAllocated.get(i);
    if(p == null) continue;
    int inputPortIndex = inputPorts.indexOf(p.getSourcePort());
    if(alreadyAllocatedInputPorts.contains(inputPortIndex)){
        outputPortPacketAllocated.set(i, null);
    } else {
        if(i>=acceptPointers.get(inputPortIndex))
            alreadyAllocatedInputPorts.add(inputPortIndex);
    }
}
```

- **Request Phase -** The requests which cannot be satisfied in same slot are removed.

```
moreIteration = false;
for(int i = 0;i<portCount;i++){
    if(alreadyAllocatedInputPorts.contains(i)){
        requestLists.get(i).clear();
    }
    Iterator<Packet> packetIterator = requestLists.get(i).iterator();

    while(packetIterator.hasNext()) {
        Packet p = packetIterator.next();
        int outputPortIndex = outputPorts.indexOf(p.getDestinationPort());
        if(alreadyAllocatedOutputPorts.contains(outputPortIndex)){
            packetIterator.remove();
        }
    }
    if(requestLists.get(i).size()>0)
        moreIteration = true;
}
```

If after all three phases, more requests can be satisfied, then more iterations are performed according to the **moreIteration** variable.

Packet Transmission

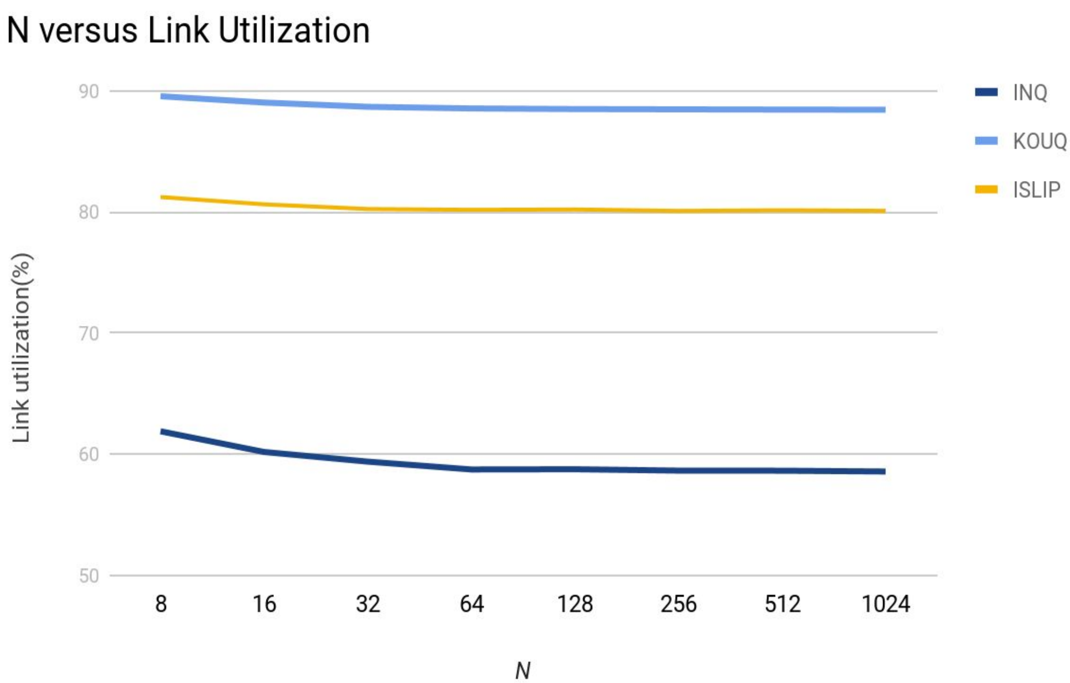
```
//Phase 3: Corresponds to packet transmission.
//Do necessary calculations here
for(OutputPort outputPort: outputPorts){
    if(outputPort.isBufferEmpty()) continue;
    Packet packet = outputPort.getPacketAtHead();

    int currentPacketDelay = time - packet.getArrivalTime();
    totalPacketDelay+= currentPacketDelay;
    totalSquarePacketDelay+= currentPacketDelay*currentPacketDelay;
    transmittedPacketCounts++;

    outputPort.removeFromBuffer();
}
```

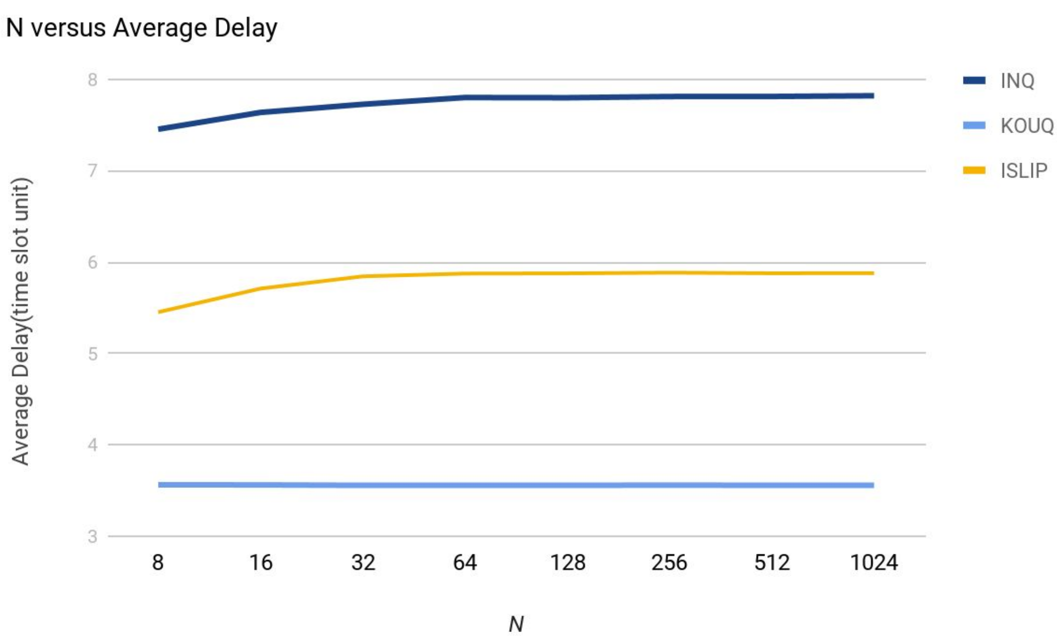
In this phase, the packet is transmitted from the top of the head of the output port.

RESULTS AND GRAPHS

Varying N (B = 4, K = 4)

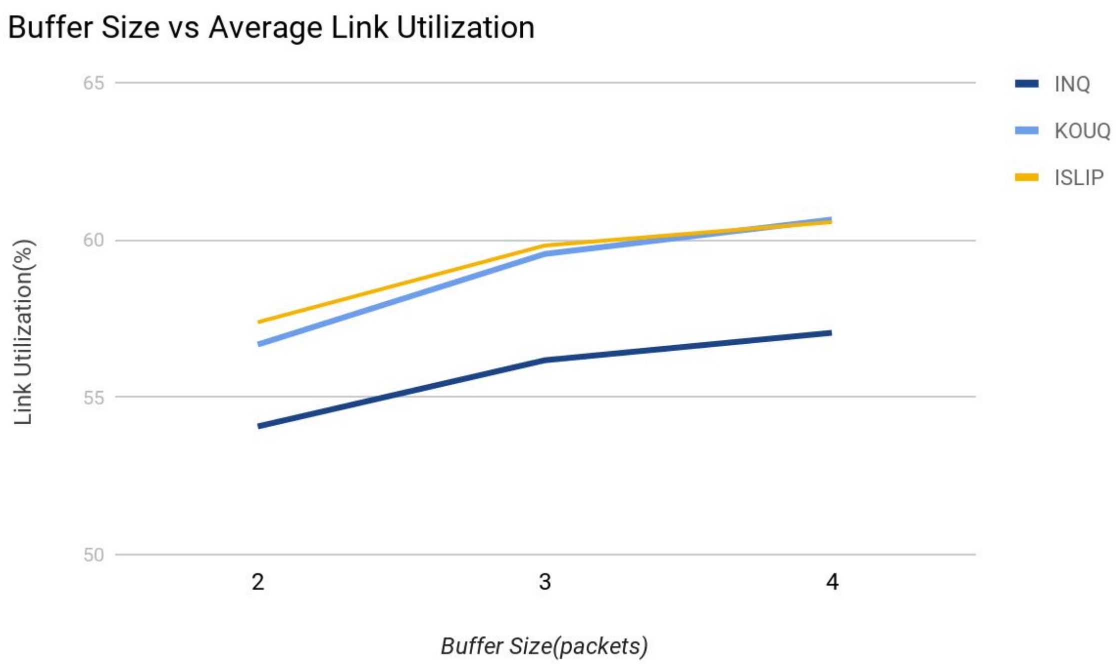
MaxTimeSlots = 10000, Packet Generation Probability = 1.

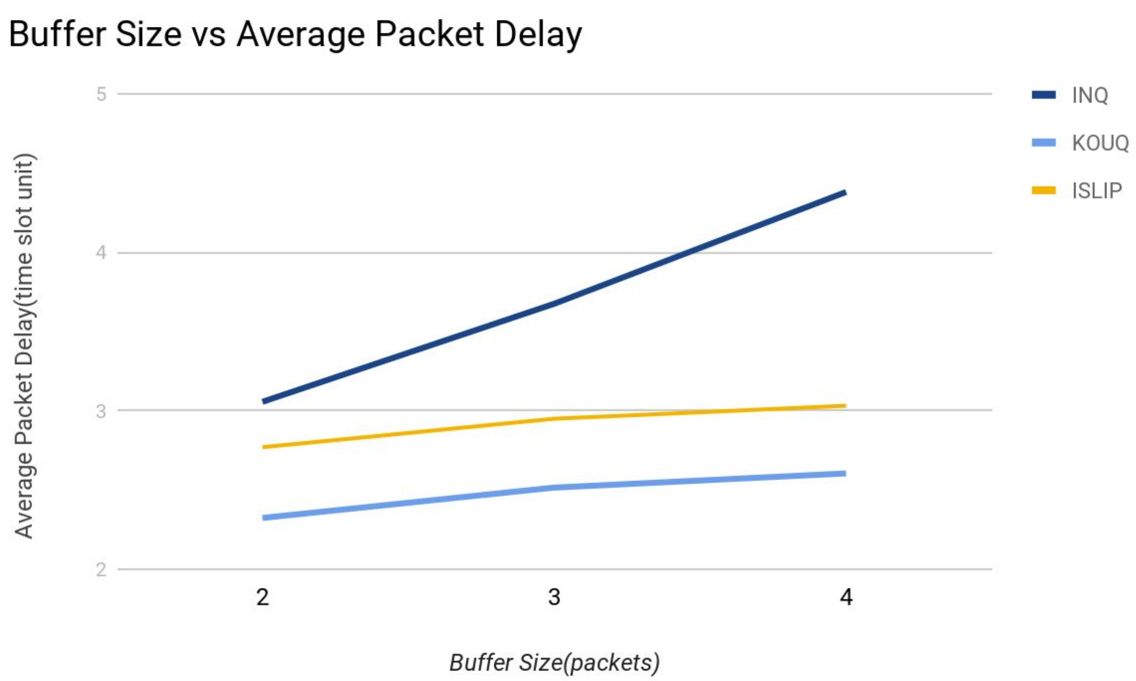




Varying B(N = 8, K = 4)

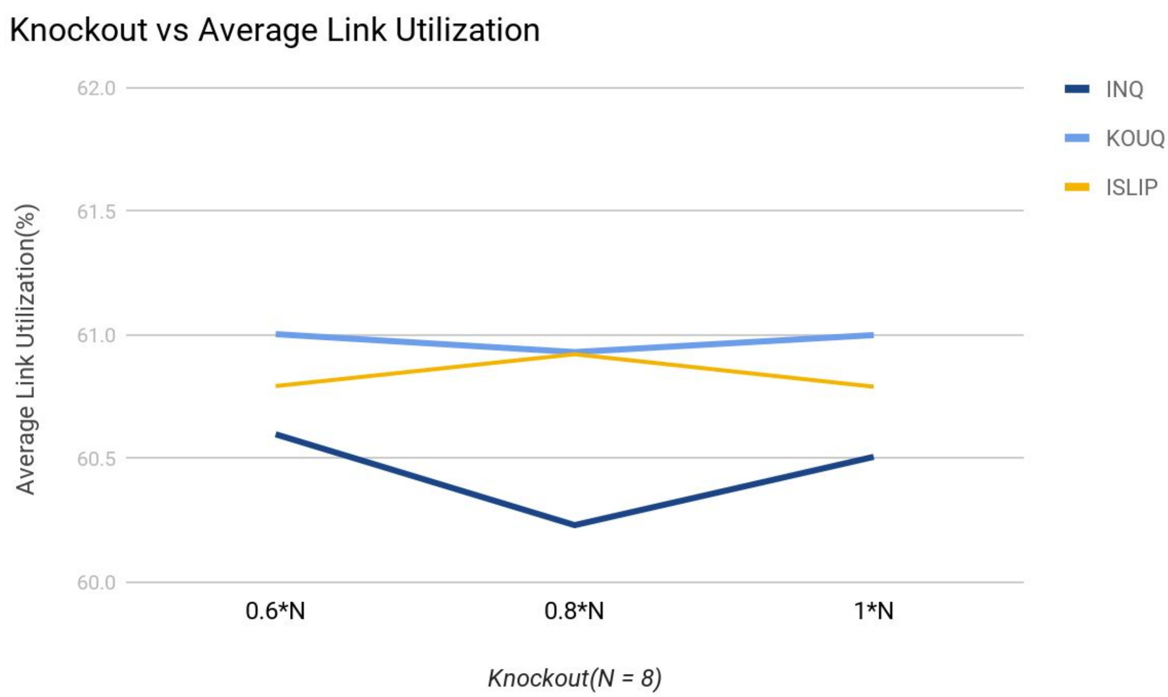
MaxTimeSlots = 10000, Packet Generation Probability = 0.6

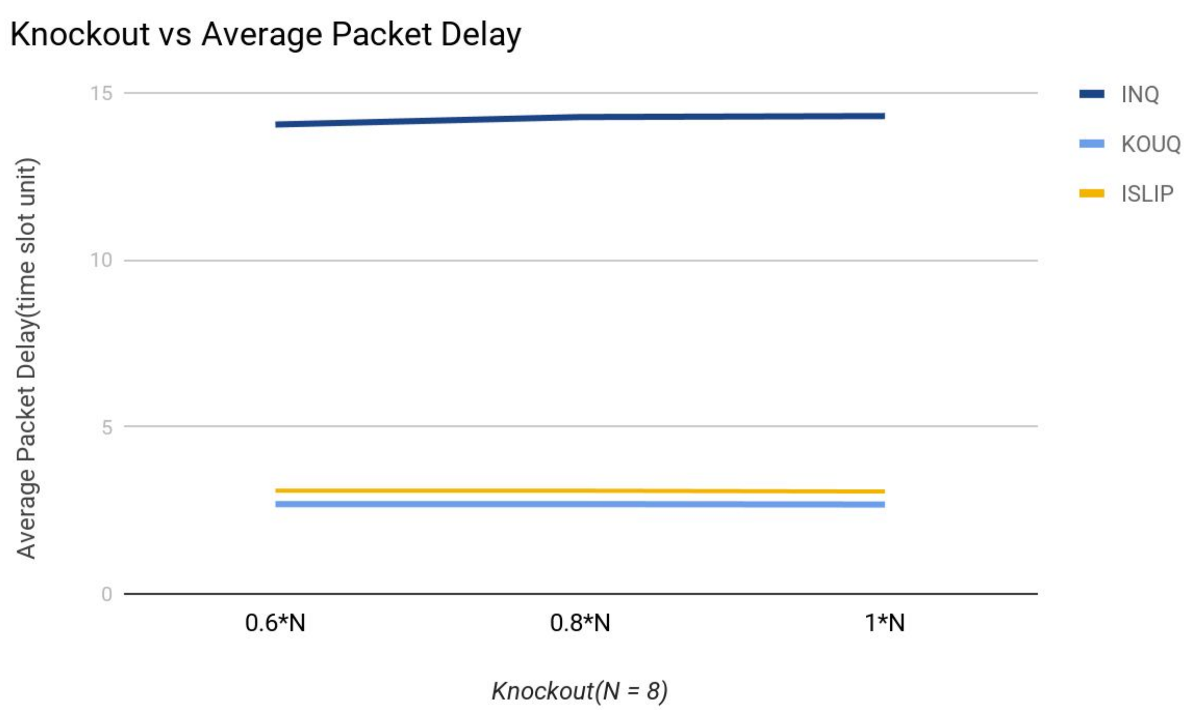




Varying K(N = 8, B = 4)

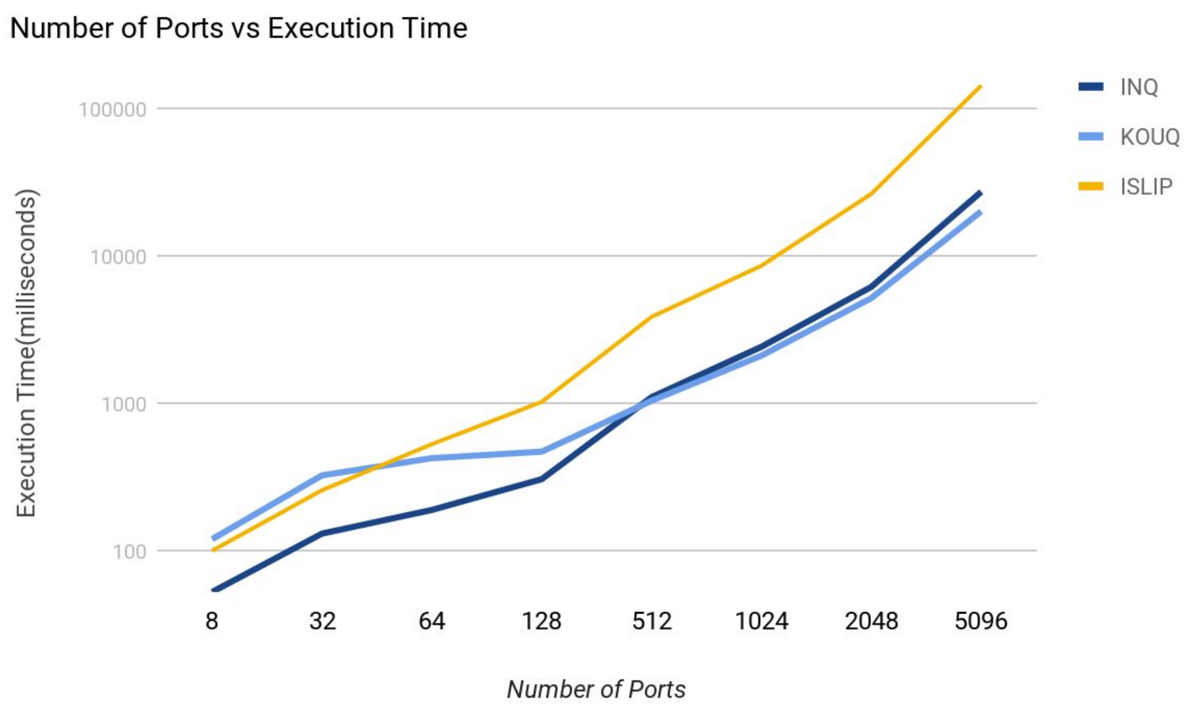
MaxTimeSlots = 10000, Packet Generation Probability = 0.6





CONCLUSION

Performance



The execution time of the program goes beyond **1 minute** for ISLIP and N = 5096.

Otherwise, it is fairly fast, as you can visualize from the graph.

Results

As evident from the graphs, **KOUQ and ISLIP** provide a sufficiently large average link utilization value.

While the **delay is minimum in KOUQ**, this is because the packets are either scheduled and transmitted after they are generated, or they are dropped, there is no buffering in the input queue (which contributes a relevant amount to waiting time).

In ISLIP, the delay is more than KOUQ but less than INQ, this is **only possible if scheduling can be done in one time slot** (which we have assumed in our program). But in practical cases, this may not be possible as scheduling takes more than one iteration and can contribute to packet delay.