Router Scheduling Algorithms: Simulation & Comparision in C++

Adarsh Gupta (220101003), Tanvi Doshi (220101102)
Vasudha Meena (220101108), Yash Jain (220101115)

CS342 Fall 2024 Assignment 4

Contents

1 Introduction

This document contains a step-by-step explanation of the implementation and the comparative study of various scheduling algorithms that can be used in the router switching fabric. The problem statement is stated below in brief.

The goal is to design and implement a network router switch fabric that handles high-throughput traffic with 8 input and output ports. The scheduling algorithm determines which input port packet should be sent to its corresponding output port. This assignment aims to explore and compare the performance of different scheduling algorithms used in the output queuing of router switch fabrics. We had to implement and analyze four scheduling algorithms: Priority Scheduling, Weighted Fair Queuing (WFQ), Round Robin (RR), and iSLIP. Each input port has variable packet arrival rate and variable priority packets. Each input and output buffer has fixed capacity of 64 packets. The metrics of comparison of each scheduling algorithm are: Queue Throughput, Turnaround Time, Waiting Time, Buffer Occupancy, and Packet Drop Rate.

The rest of the report is organized as follows, Section 2 contains the General Router Behaviour and the breakdown of the various aspects of the router functionality. Section 3 contains the implementation of each of the router functionalities as detailed in the previous section. Section 4 contains the implementation details of the various scheduling algorithms, and Section 5 contains the performance comparison of the various scheduling algorithms in the 3 simulations conditions that were asked. Section 6 contains the conclusion of the entire assignment. Further, the appendix contains queue buffer occupancy graphs.

Code Link:

2 Router Behaviour Overview

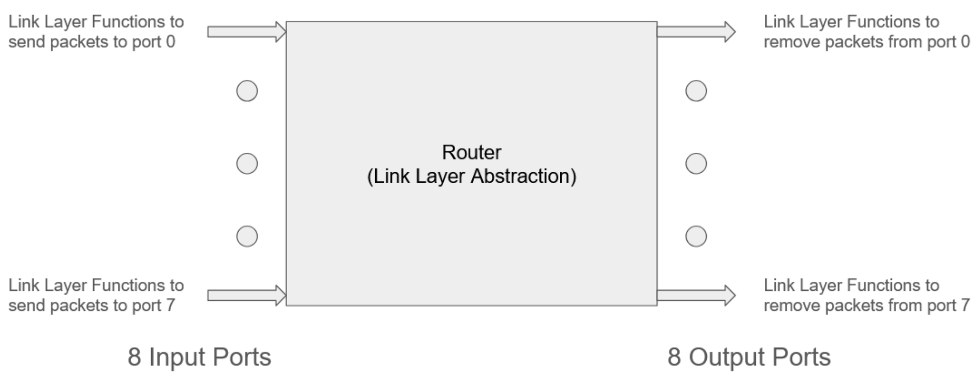

Figure 1: This figure shows the router abstraction provided to the link layer

The link layer expects the router to provide the abstraction of the following functions (This is represented in Figure 1)

```
// Link layer functions
int addToInputQueue(int, Packet*);
void removeFromOutputQueue(int);
```

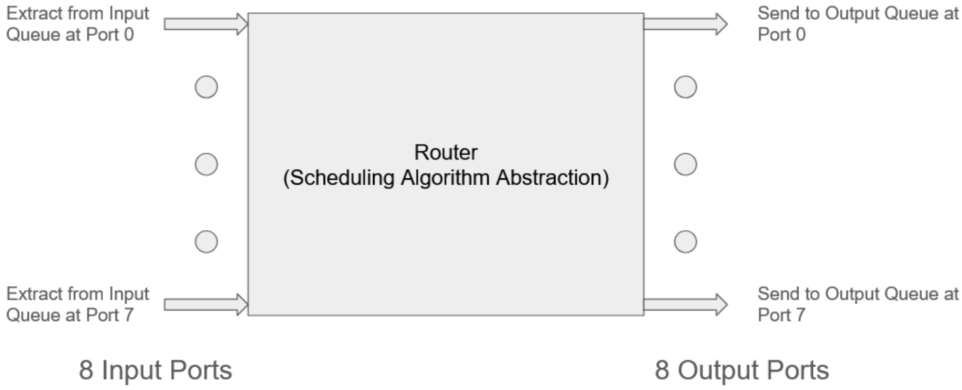

Figure 2: This figure shows the router abstraction provided to the scheduling algorithms

The scheduling algorithm expects the router to provide the abstraction of the following functions (This is represented in Figure 2)

```
    // Scheduler functions
    Packet* removeFromInputQueue(int);
    int sendToOutputQueue(int, Packet*);
    // for iSLIP need removeFromInputQueueVOQ
    Packet* removeFromInputQueueVOQ(int, int);
```

The general flow of the router for a single packet is thus as follows:

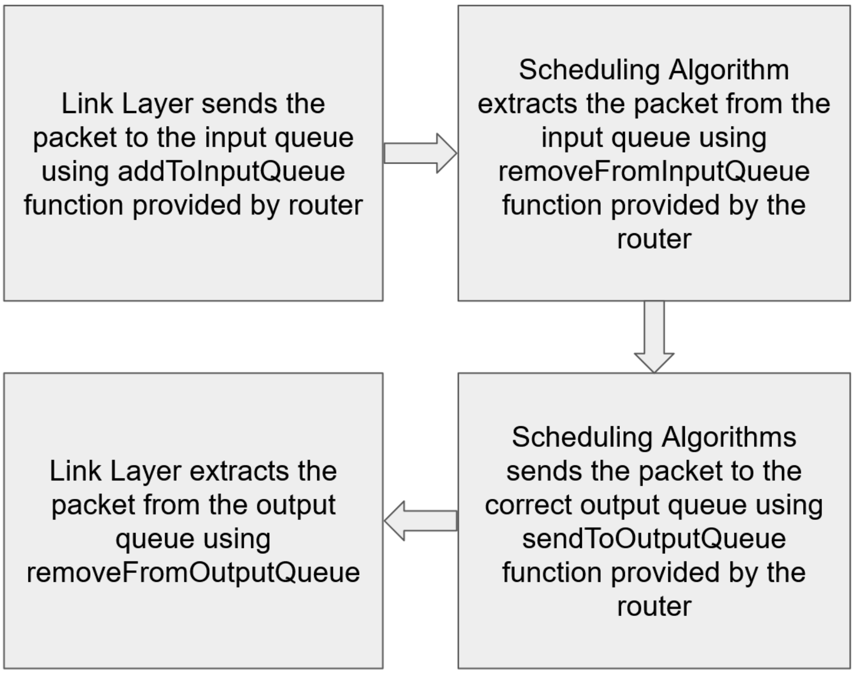

Figure 3: This figure shows the general flow of a Packet using the various abstractions provided by the router

Therefore, the general behavior of the router and link layers can be represented as follows:

```
class Router{
public:
```

```
    Buffer input[NUM_QUEUES];
    Buffer output[NUM_QUEUES];
    VOQBuffer VOQInput[NUM_QUEUES]; // needed for iSLIP algorithm

    // Link layer functions
    int addToInputQueue(int, Packet*);
    void removeFromOutputQueue(int);

    // Scheduler functions
    Packet* removeFromInputQueue(int);
    int sendToOutputQueue(int, Packet*);

    // for iSLIP need removeFromInputQueueVOQ
    Packet* removeFromInputQueueVOQ(int, int);
};

// The below 2 functions simulate the link layer
void sendToQueue(Router*, int);
void removeFromQueue(Router*, int);

// Scheduler Function
void PriorityScheduler(Router*);
void RoundRobinScheduler(Router*);
void WeightedFairScheduler(Router*);
void iSLIPScheduler(Router*);
```

3 Router Implementation

In this section, we will cover the implementation aspects of all the functionalities in the router in detail. We will first start with the Packet abstraction, then Buffer and similarly VOQ Buffer abstractions and implementation, then we will cover the Router abstraction and implementation, and finally, some utility function and other global variables (such as mutex, etc.) which control various aspects of the flow.

3.1 Packet

The Packet class is defined as follows:

```
class Packet{
public:
    int id;
    int priority;
    float arrivalTime;
    float startProcessingTime = 0; // when switching fabric processes this packet
    float sentTime = 0;         // when finally sent on output link
    // startProcessingTime is 0 for packets dropped on input queue
    // sentTime is 0 for packets dropped on output queue
    int inputPort;
    int outputPort;      // Forwarding Table, what's that?

    Packet(int id, int priority, float arrivalTime, int inputPort, int outputPort){
        this->id = id;
        this->priority = priority;
        this->arrivalTime = arrivalTime;
        this->inputPort = inputPort;
        this->outputPort = outputPort;
    }

    Packet* clone(){
        return new Packet(this->id, this->priority, this->arrivalTime, this->inputPort, this->outputPort);
    }
};
```

The id field is a unique id of each packet, controlled by a global variable pid and its corresponding mutex. The priority field is used to differentiate between low and high-priority packets (0 is high priority and 1 is low priority). The arrivalTime is the relative time (in ms) from the start of the program at which the packet is created and added to the input queue by the link layer function sendToQueue. The startProcessingTime field is the relative time at which the packet gets extracted from the inputQueue by the scheduler for processing. The sentTime is the relative time (in ms) at which the packet is removed from the output queue by the link layer function removeFromQueue. The class also provides the default constructor, keeps all the variables public to be accessed by removeFromInputQueue and removeFromQueue which populate the startProcessingTime and sentTime fields respectively.

3.2 Buffer

Each input and output queue is just an object of the Buffer Class. The buffer class is defined as follows:

```
class Buffer{
    int capacity = BUFFER_CAPACITY; // 64
    int size = 0;
    std::queue<Packet*> bufferQueue;

public:
    int push(Packet*);
    void pop();
    Packet* front();
    inline bool full() const { return size == capacity; }
    inline bool empty() const { return size == 0; }
    inline int getSize() const { return size; }
};
```

This buffer abstraction is used by the Router in both the scheduler and link layer functions. Capacity represents the maximum capacity (provided as 64 in the assignment description), size represents the current size of the queue, bufferQueue is a standard queue of Packet pointers. The push, pop and front abstraction is used by the Router functions. The implementation of these is shown below.

```
// Buffer push functions
int Buffer::push(Packet* pkt){
    if(this->full()){ return 0;} // Signifies packet dropping
    this->bufferQueue.push(pkt);
    this->size += 1;
    return pkt->id;
}

// Buffer pop function
void Buffer::pop(){
    this->bufferQueue.pop();
    this->size -= 1;
}

// Buffer front function
Packet* Buffer::front(){
    return this->bufferQueue.front();
}
```

3.3 VOQBuffer

VOQ stands for Virtual Output Queuing. This abstraction of the input queue is used by the iSLIP algorithm. Each input queue is further divided into 8 sub-input queues, each of which are identified by an outputIndex. Each sub-input queue is responsible for 1 output queue. This abstraction is used to mitigate Head of Line Blocking when using cross bar switches.

```
class VOQBuffer{
    int capacity = BUFFER_CAPACITY/NUM_QUEUES;
```

```
    int size = 0;
    std::mutex sizeMutex;
    std::queue<Packet*> bufferQueue[NUM_QUEUES];

public:
    int push(Packet*, int);
    void pop(int);
    Packet* front(int);
    inline bool full() const { return size == capacity; }
    inline bool empty() const { return size == 0; }
    inline bool empty(int i) const { return bufferQueue[i].empty(); }
    inline int getSize() const { return size; }
    inline int getSize(int i) const { return bufferQueue[i].size(); }
};
```

Essentially each input queue, is further an array of 8 queues, each containing packets of a particular output queue. The push, pop and front versions are modified to take an extra integer argument signifying the index of the sub-input queue (this is same as the output port in the packet).

```
// Virtual Output Queueing (VOQ) Buffer push function
int VOQBuffer::push(Packet* pkt, int outputIndex){
    if(this->full()){ return 0; } // Packet dropping for this input port
    this->bufferQueue[outputIndex].push(pkt);
    this->sizeMutex.lock();
    this->size++;
    this->sizeMutex.unlock();
    return pkt->id;
}

// Virtual Output Queueing (VOQ) Buffer pop function
void VOQBuffer::pop(int outputIndex){
    this->bufferQueue[outputIndex].pop();
    this->sizeMutex.lock();
    this->size--;
    this->sizeMutex.unlock();
}

// Virtual Output Queueing (VOQ) Buffer front function
Packet* VOQBuffer::front(int outputIndex){
    return this->bufferQueue[outputIndex].front();
}
```

3.4 Router

The Router class which provided abstractions to the Link layer and the Scheduler is defined as below:

```
class Router{
public:
    Buffer input[NUM_QUEUES];
    Buffer output[NUM_QUEUES];
    VOQBuffer VOQInput[NUM_QUEUES]; // needed for iSLIP algorithm

    // Link layer functions
    int addToInputQueue(int, Packet*);
    void removeFromOutputQueue(int);

    // Scheduler functions
    Packet* removeFromInputQueue(int);
    int sendToOutputQueue(int, Packet*);

    // for iSLIP need removeFromInputQueueVOQ
    Packet* removeFromInputQueueVOQ(int, int);
};
```

The input, output and VOQInput buffers are just arrays of objects of the buffer and VOQBuffer classes, which represent the input and output queues.

3.4.1 Link Layer Abstraction

The abstraction provided by the router to the link layer consists of addToInputQueue and removeFromOutputQueue. (The detailed line by line commented code of these abstractions is given below)

```
// Router Function to add to input queue (used by link layer adder)
int Router::addToInputQueue(int inputQueueNumber, Packet* pkt){
    // for iSLIP algorithm we need to use VOQ queues instead
    if(scheduler_choice == 4){
        // acquire locks on the correct sub-input queue
        inputMutexVOQ[inputQueueNumber][pkt->outputPort].lock();
        // push to that sub-input queue
        int ret = this->VOQInput[inputQueueNumber].push(pkt, pkt->outputPort);
        // unlock locks on the sub-input queue
        inputMutexVOQ[inputQueueNumber][pkt->outputPort].unlock();
        // if return value of push is 0, this signifies that the input queue is full, and thus this packet was dropped
        if(ret == 0){
            // in which case we neeed to print information for later metric calculation and delete this dynamically allocated object
            consoleMutex.lock();
            cout << *pkt;
            consoleMutex.unlock();
            delete pkt;
        }
        return ret;
    }
    // for other schedulers, normally acquire input queue lock
    inputMutex[inputQueueNumber].lock();
    // push packet to the input queue
    int ret = this->input[inputQueueNumber].push(pkt);
    // unlock the lock on the input queue
    inputMutex[inputQueueNumber].unlock();
    // similar to above, if return value is 0 then queue was full and packet was dropped
    if(ret == 0){
        // if packet was dropped, we need to print its information for tracking
        consoleMutex.lock();
        cout << *pkt;
        consoleMutex.unlock();
        delete pkt;
    }
    return ret;
}


// Router function to remove from output queue (used by link layer remover function to cleanup output queues)
void Router::removeFromOutputQueue(int outputQueueNumber){
    // go into busy wait if this output queue is empty
    while(this->output[outputQueueNumber].empty());

    // get output queue lock
    outputMutex[outputQueueNumber].lock();
    // get packet at front of the queue and remove it
    Packet* pkt = this->output[outputQueueNumber].front();
    this->output[outputQueueNumber].pop();
    // this pkt was stored at creation in the allPackets map (seen later in link layer function sendToQueue)
    map<int, Packet*>::iterator it = allPackets.find(pkt->id);
    if(it != allPackets.end()){
        // set the sentTime to the current relative time in ms
        it->second->sentTime = getTime();

        // to avoid unnecessary storage of transmitted packets, we can print the information of the packet in a file and delete it
        // this printed information can later be used for metric calculation
```

```
        consoleMutex.lock();
        cout << *it->second;
        consoleMutex.unlock();
        Packet* pktptr = it->second;
        allPackets.erase(it);
        delete pktptr;
    }

    outputMutex[outputQueueNumber].unlock();
}
```

3.4.2 Scheduler Abstraction

The abstraction provided by the router to the Scheduler consists of removeFromInputQueue, removeFromInputQueueVOQ and sendToOutputQueue. Their line by line commented code is given below:

```
// Router function to remove from input queue (used by scheduler to remove from any selected input queue for processing)
Packet* Router::removeFromInputQueue(int inputQueueNumber){
    // go into busy waiting if the input queue is empty
    while(this->input[inputQueueNumber].empty());

    // acquire input queue lock
    inputMutex[inputQueueNumber].lock();
    // get packet from input queue front
    Packet* pkt = this->input[inputQueueNumber].front();
    this->input[inputQueueNumber].pop();
    // set the startProcessingTime field in the packet
    if(allPackets.find(pkt->id) != allPackets.end()){
        allPackets.find(pkt->id)->second->startProcessingTime = getTime();
    }
    // release input queue lock
    inputMutex[inputQueueNumber].unlock();

    // return pkt pointer to be used by the scheduler
    return pkt;
}

// Router function to reomve from input queue, modified for iSLIP since VOQ is used
Packet* Router::removeFromInputQueueVOQ(int inputQueueNumber, int outputIndex){
    // busy waiting if that sub-input queue is empty
    while(this->VOQInput[inputQueueNumber].empty(outputIndex));

    // get the sub-input queue mutex lock
    inputMutexVOQ[inputQueueNumber][outputIndex].lock();
    // extract packet from sub-input queue
    Packet* pkt = this->VOQInput[inputQueueNumber].front(outputIndex);
    this->VOQInput[inputQueueNumber].pop(outputIndex);
    // set the startProcessingTime for the packet
    if(allPackets.find(pkt->id) != allPackets.end()){
        allPackets.find(pkt->id)->second->startProcessingTime = getTime();
    }
    // release sub-input queue mutex lock
    inputMutexVOQ[inputQueueNumber][outputIndex].unlock();

    // return pkt pointer to be used by the scheduler
    return pkt;
}

// Router function to send to output queue (used by scheduler to send packet to correct output queue)
int Router::sendToOutputQueue(int outputQueueNumber, Packet* pkt){
    // acquire output queue lock
    outputMutex[outputQueueNumber].lock();
    // push packet to output queue
    int ret = this->output[outputQueueNumber].push(pkt);
    // release output queue lock
```

```
    outputMutex[outputQueueNumber].unlock();
    return ret;
}
```

3.5 Link Layer Functions

The link layer uses the addToInputQueue and removeFromOutputQueue abstractions provided by the routers and implement continous sending of packets to each of the input queue and continous removal of the packets from the output queues.

3.5.1 sendToQueue

This (I agree slightly complicated) function implements the general packet sending logic for each input queue. This function is started and detached on seperate threads for each of the input queues. (The logic of this function was manipulated when simulating different behaviours example constant traffic for all queues, different for different queues etc., but that change is just changing the external if conditions and not the core logic of bursty and uniform traffic)

```
// Link Layer Adding Function
void sendToQueue(Router * router, int inputQueueNumber){
    // This stop_threads is an atomic variable which is set to True after the simulation time is over
    while(!stop_threads.load()){
        // The sleep pattern and priority will depend on the queue number
        if((inputQueueNumber == 0) || (inputQueueNumber == 4) || (inputQueueNumber == 2) || (inputQueueNumber == 6) ){
        // Bursty Traffic
            // Sample Number of packets to be sent in burst
            int numPackets = BURST_LOW + rand()%(BURST_HIGH - BURST_LOW + 1);

            int priority;
            if((inputQueueNumber == 0) || (inputQueueNumber == 4)){
                // High priority traffic
                priority = 0; // (lower is higher)
            }else if((inputQueueNumber == 2) || (inputQueueNumber == 6)){
                // Low priority traffic
                priority = 1;
            }

            // Allocate that many PIDs
            pidMutex.lock();
            int basePID = pid + 1;
            int maxPID = pid + numPackets;
            pid += numPackets;
            pidMutex.unlock();

            // Create & Send numPackets packets
            for(int i = basePID; i <= maxPID; i++){
                // Dynamically Allocates Packet Object
                Packet* currPacket = new Packet(i, priority, getTime(), inputQueueNumber, rand()%8);
                // add the packet to the allPackets map
                allPackets.emplace(i, currPacket);

                // Push to Router
                router->addToInputQueue(inputQueueNumber, currPacket);

                // Sleep for 3-5 ms before sending another packet to this queue to simulate the burst
                this_thread::sleep_for(chrono::milliseconds(3 + ( std::rand() % ( 5 - 3 + 1 ) )));
            }

            // Sleep for 2.5-4 seconds between bursts
            this_thread::sleep_for(chrono::milliseconds(2500 + ( std::rand() % ( 4000 - 2500 + 1 ) )));
        }else if((inputQueueNumber == 1) || (inputQueueNumber == 5) || (inputQueueNumber == 3) || (inputQueueNumber == 7)){
        // Uniform Traffic
```

```
            pidMutex.lock();
            int currPID = ++pid;
            pidMutex.unlock();

            int priority;
            if((inputQueueNumber == 1) || (inputQueueNumber == 5)){
                // High priority traffic
                priority = 0; // (lower is higher)
            }else if((inputQueueNumber == 3) || (inputQueueNumber == 7)){
                // Low priority traffic
                priority = 1;
            }

            // dynamically allocate new packet
            Packet* currPacket = new Packet(currPID, priority, getTime(), inputQueueNumber, rand()%8);
            // add the packet to the allPackets map
            allPackets.emplace(currPID, currPacket);

            // push to router
            router->addToInputQueue(inputQueueNumber, currPacket);

            // Sleep for 40-80 ms to simulate constant traffic
            this_thread::sleep_for(chrono::milliseconds(40 + rand()%(80 - 40 + 1)));
        }
    }
}
```

3.5.2 removeFromQueue

This function simulates the outgoing link layer actions by constantly removing packets from each of the output queues. This function is started and detached on seperate thread in the main function for each of the output queues.

```
// Link Layer Removing Function
void removeFromQueue(Router * router, int outputQueueNumber){
    while(!stop_threads.load()){
        router->removeFromOutputQueue(outputQueueNumber);
        this_thread::sleep_for(chrono::milliseconds(75));
    }
}
```

3.6 The Main function and Global Variables

The main function is responsible for doing the following:

1. Instantiating a new Router object
2. Getting user input for selected scheduler
3. Starting detached thread for simulating scheduler on router
4. Start SizeThread which tracks each buffer occupancy over time
5. Start detached threads for simulating sendToQueue for each input port
6. Start detached threads for simulating removeFromQueue for each output port
7. End the simulation
8. Starting child process which calculates and stores the metrics report

```cpp
// Global Variables
mutex inputMutex[NUM_QUEUES];                       // Input queue mutexes
mutex inputMutexVOQ[NUM_QUEUES][NUM_QUEUES];        // used for VOQ required in iSLIP algorithm
mutex outputMutex[NUM_QUEUES];                      // Output queue mutexes
std::atomic<bool> stop_threads(false);              // For stopping all threads after simulationt time is finished
int pid = 0;                                        // packet id's
mutex pidMutex;                                     // mutex over PID
map<int, Packet*> allPackets;                       // map to maintain all dynamically allocated packets
std::chrono::system_clock::time_point startTime;// startTime is used by getTime() for populating time values in packets
int scheduler_choice;                               // global scheduler choice used to take scheduler specific actions, if any
mutex consoleMutex;                                 // console mutex ig understandable
string queueFileName;                               // File name to store buffer tracking data

int main(int argc, char* argv[]){
    startTime = std::chrono::system_clock::now(); // Set the startTime variable to the current time

    Router *router = new Router; // Instantiate Router

    // Create & Detach appropriate Scheduler thread based on user input
    cout << "Select Scheduler:\n1. Priority Scheduler\n2. Weighted Fair Scheduler\n3. Round Robin Scheduler\n4. iSLIP Scheduler\n";
    cin >> scheduler_choice;
    string filename;
    if(scheduler_choice == 1){
        thread scheduler(PriorityScheduler, router);
        scheduler.detach();
        filename = "./SimulationOutput/PriorityScheduler.txt";
        queueFileName = "./SimulationOutput/PrioritySchedulerQueue.txt";
    }else if(scheduler_choice == 2){
        thread scheduler(WeightedFairScheduler, router);
        scheduler.detach();
        filename = "./SimulationOutput/WeightedFairScheduler.txt";
        queueFileName = "./SimulationOutput/WeightedFairSchedulerQueue.txt";
    }else if(scheduler_choice == 3){
        thread scheduler(RoundRobinScheduler, router);
        scheduler.detach();
        filename = "./SimulationOutput/RoundRobinScheduler.txt";
        queueFileName = "./SimulationOutput/RoundRobinSchedulerQueue.txt";
    }else if(scheduler_choice == 4){
        thread scheduler(iSLIPScheduler, router);
        scheduler.detach();
        filename = "./SimulationOutput/iSLIPScheduler.txt";
        queueFileName = "./SimulationOutput/iSLIPSchedulerQueue.txt";
    }else{
        cout << "Invalid Choice. Exiting...\n";
        return 1;
    }

    // Change output file to store packet data
    createFoler("SimulationOutput/");
    createFoler("SimulationReports/");
    freopen(filename.c_str(), "w", stdout);

    // Create & Detach Thread for Tracking Size of each queue
    thread SizeThread(trackSize, router);
    SizeThread.detach();

    // START Link Layer Functions:
    // Start detached threads for inputting to queues
    vector<thread> InputDataSenderThreads;
    for(int i = 0; i < NUM_QUEUES; i++)
        InputDataSenderThreads.push_back(thread(sendToQueue, router, i));
    for(int i = 0; i < NUM_QUEUES; i++)
        InputDataSenderThreads[i].detach();
    // Start detached threads for removing data from output queues
    vector<thread> OutputQueueRemover;
```

```
    for(int i = 0; i < NUM_QUEUES; i++)
        OutputQueueRemover.push_back(thread(removeFromQueue, router, i));
    for(int i = 0; i < NUM_QUEUES; i++)
        OutputQueueRemover[i].detach();

    // Sleep for some time and then kill all threads
    std::this_thread::sleep_for(std::chrono::seconds(SIMULATION_TIME));
    stop_threads.store(true);
    std::this_thread::sleep_for(std::chrono::seconds(3));

    // Change back the stdout
    #ifdef _WIN32
        freopen("CON", "w", stdout);  // For Windows
    #else
        freopen("/dev/tty", "w", stdout); // For Linux/Mac
    #endif

    cout << "DONE SIMULATION!\n";
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Get Metrics
    int pid = fork();
    if(pid == 0){
        // Child process: execute ./metrics
        execl("./metrics", "./metrics", to_string(scheduler_choice).c_str(), (char *) nullptr);
        return 1;
    }else{
        // Parent process: wait for the child process to finish
        int status;
        waitpid(pid, &status, 0);  // Wait for the child
    }

    return 0;
}
```

3.7 Metric Calculation

The data for all the packets is outputted in the SimulationResults/[AlgorithmName].txt. The metrics calculator program is started by the main router program and it calculates all the metrics of the simulation by reading this particular file. The metrics.cpp file contains the code of the metrics program, and it starts by reading the file in which all the simulation data is saved. Each line of the simulation data is of the following format:

```
PacketId Priority InputPort OutputPort ArrivalTime StartProcessingTime SentTime
```

The metrics program iterates through the entire file and extracts the following information:

```
    int totalPackets = 0;
    int totalSuccessfullyTransmitted = 0;
    int totalDropped = 0;
    int queuewiseTotalPackets[NUM_QUEUES] = {};
    int queuewiseTotalSuccessfulPackets[NUM_QUEUES] = {};
    int queuewiseTotalDroppedPackets[NUM_QUEUES] = {};
    float totalWaitingTime = 0;
    float queuewiseTotalWaitingTime[NUM_QUEUES] = {};
    float totalTurnaroundTime = 0;
    float queuewiseTotalTurnaroundTime[NUM_QUEUES] = {};
```

using the following calculation for each packet:

```
            totalPackets++;
            queuewiseTotalPackets[inputPort]++;
            if(sentTime != 0){ // Not a dropped packet
                queuewiseTotalSuccessfulPackets[inputPort]++;
```

```
            totalSuccessfullyTransmitted++;

            totalTurnaroundTime += sentTime - arrivalTime;
            queuewiseTotalTurnaroundTime[inputPort] += sentTime - arrivalTime;

            totalWaitingTime += startProcessingTime - arrivalTime;
            queuewiseTotalWaitingTime[inputPort] += startProcessingTime - arrivalTime;
        }else{
            queuewiseTotalDroppedPackets[inputPort]++;
            totalDropped++;
        }
```

This information is then used to calculate the metrics as detailed below:

3.7.1 Queue Throughput

$$\text{Overall Queue Throughput} = \frac{\text{Total Packets Successfully Transmitted}}{\text{Simulation Time}}$$

$$\text{Queue Throughput}_i = \frac{\text{Total Input Packets for this Queue Successfully Transmitted}}{\text{Simulation Time}}$$

where Queue Throughput$_i$ is the queue throughput of the i^{th} input queue. From an implementation point of view, this is printed in the report as

```
cout << "\n== Queue Throughput ==\n";
cout << "Combined Router Throughput: " << totalSuccessfullyTransmitted/(float)SIMULATION_TIME << "
 Packets/Second \n";
cout << "Queue Throughput for each Input Queue: \n";
for(int i = 0; i < NUM_QUEUES; i++){
    cout << "Queue " << i << ": " << queuewiseTotalSuccessfulPackets[i]/(float)SIMULATION_TIME << "
 Packets/Second \n";
}
```

3.7.2 Turnaround Time

$$\text{Average Turnaround Time} = \frac{\text{Total Turnaround Time}}{\text{Number of packets successfully transmitted}}$$

where

$$\text{Total Turnaround Time} = \sum_{\text{all packets}} \text{Sent Time - Arrival Time}$$

$$\text{Turnaround Time}_i = \frac{\text{Total Turnaround time for this queue packets}}{\text{Successfully transmitted packets on this input queue}}$$

where Turnaround Time$_i$ is the total turnaround time (sent time - arrival time) for packets arriving on this input queue. From an implementation point of view, this is printed in the report as

```
// Turnaround Time
cout << "\n== Turnaround Time ==\n";
cout << "Average Turnaround Time: " << totalTurnaroundTime/totalSuccessfullyTransmitted << " ms \n";
cout << "Turnaround Time for Each Input Queue: \n";
for(int i = 0; i < NUM_QUEUES; i++){
    cout << "Queue " << i << ": " << queuewiseTotalTurnaroundTime[i]/queuewiseTotalSuccessfulPackets[i]
 << " ms \n";
}
```

3.7.3 Waiting Time

$$\text{Average Waiting Time} = \frac{\text{Total Waiting Time}}{\text{Number of packets successfully transmitted}}$$

where

$$\text{Total Waiting Time} = \sum_{\text{all packets}} \text{Start Processing Time - Arrival Time}$$

$$\text{Waiting Time}_i = \frac{\text{Total Waiting time for this queue packets}}{\text{Successfully transmitted packets on this input queue}}$$

where Waiting Time_i is the total waiting time (start processing time - arrival time) for packets arriving on this input queue. From an implementation point of view, this is printed in the report as

```
    // Waiting Time
    cout << "\n== Waiting Time ==\n";
    cout << "Average Waiting Time: " << totalWaitingTime/totalSuccessfullyTransmitted << " ms \n";
    cout << "Waiting Time for Each Input Queue: \n";
    for(int i = 0; i < NUM_QUEUES; i++){
        cout << "Queue " << i << ": " << queuewiseTotalWaitingTime[i]/queuewiseTotalSuccessfulPackets[i] << "
     ms \n";
    }
```

3.7.4 Buffer Occupancy

The occupancy of each input queue and output queue is tracked throughout the simulation by a seperate thread called SizeThread, which runs the trackSize function and stores the result in a file named SimulationOutput/[AlgorithmName]Queue.txt

```
// trackSize runs on a seperate thread and tracks input and output buffer Occupancy
void trackSize(Router* router){
    // Vector of Input and Output Queue Sizes
    std::vector<std::vector<int>> inputSizes;
    std::vector<std::vector<int>> outputSizes;

    std::ofstream outFile(queueFileName); // Output file stream

    // Keep going until simulation ends
    while (!stop_threads.load()) {
        // Record the current sizes
        std::vector<int> currentInputSizes;
        std::vector<int> currentOutputSizes;

        // Record Input Queue Sizes by calling appropriate buffer or VOQBuffer functions
        for (int i = 0; i < NUM_QUEUES; i++) {
            if(scheduler_choice == 4){
                currentInputSizes.push_back(router->VOQInput[i].getSize());
            }else{
                currentInputSizes.push_back(router->input[i].getSize());
            }
        }

        // Record Output Queue Sized by calling appropriate Buffer function
        for (int i = 0; i < NUM_QUEUES; i++) {
            currentOutputSizes.push_back(router->output[i].getSize());
        }

        // Store the sizes in a vector
        inputSizes.push_back(currentInputSizes);
        outputSizes.push_back(currentOutputSizes);

        // Store the sizes in a file with the current time
        outFile << "Time: " << getTime() << "\n";
        outFile << "Input Queue Sizes: ";
```

```cpp
        for (const auto& size : currentInputSizes) {
            outFile << size << " ";
        }
        outFile << "\n";

        outFile << "Output Queue Sizes: ";
        for (const auto& size : currentOutputSizes) {
            outFile << size << " ";
        }
        outFile << "\n\n";
        outFile.flush();

        // Sleep for 50 milliseconds before taking another recording
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
    }
    outFile.close();
}
```

3.7.5 Packet Drop Rate

$$\text{Percentage of Total Packets Dropped} = \frac{\text{Total Packets Dropped}}{\text{Total Packets Recorded}} * 100$$

$$\% \text{ Packets Dropped}_i = \frac{\text{Total Dropped Input Packets for this Queue}}{\text{Total Packets on this Queue}} * 100$$

where $\%$ Packets Dropped$_i$ is the percentage of packets arriving at the i^{th} input queue that are dropped. From an implementation point of view, this is printed in the report as

```cpp
    // Packet Drop Rate
    logfile << "\n== Packet Drop Rates ==\n";
    logfile << "Percentage of Total Packets Dropped: " << (totalDropped/(float)totalPackets)*100 << "%\n";
    logfile << "Percentage of Packets Dropped for each Input Queue: \n";
    for(int i = 0; i < NUM_QUEUES; i++){
        logfile << "Queue " << i << ": " << (queuewiseTotalDroppedPackets[i]/(float)queuewiseTotalPackets[i])*100 << "%\n";
    }
```

4 Scheduling Algorithms Implementation

The Scheduling Algorithms are implemented by using the removeFromInputQueue and sendToOutputQueue abstraction provided by the router along with their own logic. The general code outline for each scheduling algorithm is as follows:

```
while(!stop_threads.load()){
    Select Input Queue based on some logic
    Use removeFromInputQueue[VOQ] to get packet from this queue
    Sleep to simulate switching fabric delay
    Send to output queue using sendToOutputQueue
}
```

4.1 Priority Scheduling

For priority scheduling, their is an absolute priority assigned to the queues, given as follows:

$0 > 4 > 1 > 5 > 2 > 6 > 3 > 7$

because of the requirement of the assignment, which metions that some input queues receive high priority traffic while others receive low priority traffic. This priority convention is also followed while generating the traffic in sendToQueue function of the link layer. (Another convention followed is that Bursty Traffic has a higher priority than Uniform Traffic).

```
void PriorityScheduler(Router* router){
    // Queues 0, 1, 4, 5 have high priority traffic
    // Queues 2, 3, 6, 7 have low priority traffic
    while(!stop_threads.load()){
        // Based on priority
        // we need to decide which queue to take data from
        int selectedQueue;
        if(!(router->input[0].empty()))
            selectedQueue = 0;
        else if(!(router->input[4].empty()))
            selectedQueue = 4;
        else if(!(router->input[1].empty()))
            selectedQueue = 1;
        else if(!(router->input[5].empty()))
            selectedQueue = 5;
        else if(!(router->input[2].empty()))
            selectedQueue = 2;
        else if(!(router->input[6].empty()))
            selectedQueue = 6;
        else if(!(router->input[3].empty()))
            selectedQueue = 3;
        else if(!(router->input[7].empty()))
            selectedQueue = 7;
        else
            continue;

        // Get packet from Router Current Input Queue
        Packet* pkt = router->removeFromInputQueue(selectedQueue);

        // Sleep to simulate switching fabric delay
        this_thread::sleep_for(chrono::milliseconds(SWITCH_DELAY));

        // Send to Corrent Output Queue
        router->sendToOutputQueue(pkt->outputPort, pkt);
    }
}
```

4.2 Weighted Fair Scheduling

This is implemented probablistically using Lottery Scheduling. The weights assigned are as follows:

```
Queue  |  Traffic Type           |  Tickets
 0, 4  |   Bursty High Priority  |    100
 1, 5  |   Uniform High Priority |     90
 2, 6  |   Bursty Low Priority   |     50
 3, 7  |   Uniform Low Priority  |     40
```

```
void WeightedFairScheduler(Router* router){
    // WFQ Scheduler can be implemented probablistically using Lottery Scheduling
    // Setting tickets
    int tickets[NUM_QUEUES] = {};
    tickets[0] = tickets[4] = 100; // High Priority Bursty Traffic
    tickets[1] = tickets[5] = 90; // High Priority Uniform Traffic
    tickets[2] = tickets[6] = 50; // Low Priority Bursty Traffic
    tickets[3] = tickets[7] = 40; // Low Priority Uniform Traffic
    int totalTickets = 560;
    std::random_device rd;  // Non-deterministic random number generator
    std::mt19937 gen(rd()); // Seed the Mersenne Twister random number generator
    std::uniform_int_distribution<> distrib(1, totalTickets); // To sample numbers uniformly between 1, TotalTickets

```

```
    while(!(stop_threads.load())){
        // Sample winning tickets
        int winningTickets = distrib(gen);
        int currentTickets = 0;
        int selectedQueue = 0;

        // Using winning tickets get winning queue
        for(; selectedQueue < NUM_QUEUES; selectedQueue++){
            currentTickets += tickets[selectedQueue];
            if (winningTickets <= currentTickets) {
                break;
            }
        }

        // Get packet from Router Current Input Queue
        Packet* pkt = router->removeFromInputQueue(selectedQueue);

        // Sleep to simulate switching fabric delay
        this_thread::sleep_for(chrono::milliseconds(SWITCH_DELAY));

        // Send to Corrent Output Queue
        router->sendToOutputQueue(pkt->outputPort, pkt);
    }
}
```

4.3 Round Robin Scheduler

```
void RoundRobinScheduler(Router* router){
    // all it does is iterates through the input queues, takes one and sends it to the output queues with
     waits in between
    int currQueue = 0;
    while(!stop_threads.load()){
        // Get packet from Router Current Input Queue
        Packet* pkt = router->removeFromInputQueue(currQueue);

        // Sleep to simulate switching fabric delay
        this_thread::sleep_for(chrono::milliseconds(SWITCH_DELAY));

        // Send to Corrent Output Queue
        router->sendToOutputQueue(pkt->outputPort, pkt);

        // Go to next Queue
        currQueue = (currQueue+1)%8;
    }
}
```

4.4 iSLIP Scheduler

The working of the algorithm itself is quite simple, there are three phases for a single decision: Request, Grant and Accept. In the request phase each input queue makes a request to the output queues for which it has existing packets. In the grant phase, the output queues sees the list of input queues that has made a request to access it and in a round robin fashion accepts the next input queue. In the accept phase, this information is used by the input queue to finalise the output queue to which it needs to send the data.

The VOQBuffers are used to implement sub-input queues cooresponding to each output queue at every input queue.

```
void iSLIPScheduler(Router* router){
    // Round-robin pointers for inputs and outputs queuess
    std::vector<int> inputPointers(NUM_QUEUES, 0);
    std::vector<int> outputPointers(NUM_QUEUES, 0);

```

```cpp
    // While simulation goes on
    while (!stop_threads.load()) {
        std::vector<int> grantedOutputs(NUM_QUEUES, -1); // Track which output has granted to which input
        std::vector<int> acceptedInputs(NUM_QUEUES, -1); // Track which input has accepted which output

        // Step 1: Request Phase
        // Each input requests to all outputs for which it has packets queued
        std::vector<std::vector<bool>> requests(NUM_QUEUES, std::vector<bool>(NUM_QUEUES, false));
        for (int i = 0; i < NUM_QUEUES; i++) {
            for (int j = 0; j < NUM_QUEUES; j++) {
                if(!router->VOQInput[i].empty(j)){ // Important check here
                    requests[i][j] = true;
                }
            }
        }

        // Step 2: Grant Phase
        // Each output reviews the requests and grants to one input
        for (int j = 0; j < NUM_QUEUES; j++) {
            for (int i = 0; i < NUM_QUEUES; i++) {
                // the output queue iterates in a round robin fashion starting from the current input queue pointed by the output queue
                int inputIndex = (outputPointers[j] + i) % NUM_QUEUES;
                if (requests[inputIndex][j]) {
                    grantedOutputs[j] = inputIndex;  // Output j grants to inputIndex
                    break;
                }
            }
        }

        // Step 3: Accept Phase
        // Each input that received one or more grants selects one output
        for (int i = 0; i < NUM_QUEUES; i++) {
            // Round Robin for input pointers
            for (int j = 0; j < NUM_QUEUES; j++) {
                int outputIndex = (inputPointers[i] + j) % NUM_QUEUES;
                if (grantedOutputs[outputIndex] == i) {
                    acceptedInputs[i] = outputIndex;  // Input i accepts output outputIndex
                    break;
                }else{
                    acceptedInputs[i] = -1;
                }
            }
        }

        // Packet Transfer based on accepted grants
        for (int i = 0; i < NUM_QUEUES; i++) {
            if (acceptedInputs[i] != -1) {
                int selectedOutput = acceptedInputs[i];

                // Remove packet from input queue and send to output queue
                Packet* pkt = router->removeFromInputQueueVOQ(i, selectedOutput);

                if (pkt) {
                    std::this_thread::sleep_for(std::chrono::milliseconds(SWITCH_DELAY));  // Simulate switching delay
                    router->sendToOutputQueue(selectedOutput, pkt);

                    // Update the round-robin pointer only if grant was accepted by input queue, which is signified by packet transmission
                    inputPointers[i] = (inputPointers[i] + 1) % NUM_QUEUES;
                    outputPointers[selectedOutput] = (outputPointers[selectedOutput] + 1) % NUM_QUEUES;
                }

                requests[i][selectedOutput] = false;
            }
        }
    }
```

```
}
```

5 Scheduling Algorithms Comparison

Refer to the Appendix for Occupancy Graphs

Note that the times set for the various delays were very high compared to actual routers, this was done to ensure we did not produce so many packets that could not be simulated on my computer. The simulation time was also kept small, again for the same reason. The purpose of these tables is just for the comparison of the turnaround, waiting times, and queue throughput and not for the actual absolute values. The buffer occupancy graphs show continuously increasing values in some cases, which is due to the standard parameters across entire testing which had values much higher than actual, again just for testing. For queuewise data of all metrics refer to the GitHub link in the conclusion.

5.1 Uniform Traffic

Uniform Traffic means All Input Ports receive uniform traffic of the same priority.

	Priority Scheduler	**Weighted Fair Scheduler**	**Round Robin Scheduler**	**iSLIP Scheduler**
Queue Throughput	95.1 pkt/s	97.4 pkt/s	98.9 pkt/s	97.9 pkt/s
Average Turnaround Time	1616.58ms	1473.8ms	1379.32ms	1305.89ms
Average Waiting Time	1568.84ms	1429.32ms	1332.67ms	1290.26ms
Packet Drop Rate	0.10%	0%	0%	0%

Table 1: Comparison of Different Scheduling Algorithms with uniform traffic

5.2 Non-Uniform Traffic

Non-Uniform Traffic (According to the assignment) means All Input Ports receive uniform traffic of different priority.

	Priority Scheduler	**Weighted Fair Scheduler**	**Round Robin Scheduler**	**iSLIP Scheduler**
Queue Throughput	98.8 pkt/s	89.2 pkt/s	98.7 pkt/s	97.9 pkt/s
Average Turnaround Time	145.67ms	1450.44ms	1383.51ms	1299.26ms
Average Waiting Time	99.73ms	1411.1ms	1336.36ms	1283.53ms
Packet Drop Rate	17.39%	10.44%	0%	0%

Table 2: Comparison of Different Scheduling Algorithms with non-uniform traffic

5.3 Bursty Traffic

Bursty Traffic means all input ports receive either bursty or uniform traffic of different priorities. The provided and explained code in this report simulates this kind of traffic.

	Priority Scheduler	**Weighted Fair Scheduler**	**Round Robin Scheduler**	**iSLIP Scheduler**
Queue Throughput	93.7 pkt/s	89.9 pkt/s	74.6 pkt/s	96.9 pkt/s
Average Turnaround Time	1670.52ms	1632.38ms	1817.68ms	1155.84ms
Average Waiting Time	1448.52ms	1353.19ms	1776.89ms	1135.61ms
Packet Drop Rate	0.63%	10.55%	5.20966%	0%

Table 3: Comparison of Different Scheduling Algorithms with complicated bursty traffic

6 Conclusion

All the code and data are located at: https://github.com/CoolSunflower/RouterC-/
In this section, we will answer the following questions:

6.1 Which algorithm achieves the lowest packet delay and why?

The **iSLIP algorithm** achieves the lowest packet delay, particularly in bursty traffic scenarios. This is because iSLIP implements a more advanced method of allocating requests from input ports to output ports, utilizing a round-robin mechanism that prevents any single input port from being starved of access. By handling virtual output queues (VOQ) and reducing head-of-line blocking, iSLIP significantly minimizes the turnaround and waiting times compared to other algorithms.

6.2 How does each algorithm handle high-priority traffic versus low-priority traffic?

1. Priority Scheduling strictly prioritizes high-priority traffic, leading to potential starvation of low-priority packets, especially in bursty conditions. (This can be seen in the results, since for bursty traffic, the low-priority queues had high drop rate)

2. Weighted Fair Queuing (WFQ) assigns weights to different queues, balancing the service of high and low-priority traffic by giving more weight (and thus more chances) to higher-priority queues. However, it does not eliminate starvation entirely for low-priority traffic under extreme conditions. (In the simulation results, Weighted Fair Queueing does reduce the average turnaround time for low priority traffic but at the cost of higher packet drop rate of bursty traffic)

3. Round Robin (RR) treats all queues equally, regardless of traffic priority, so high-priority traffic does not receive special treatment, which could increase delay for high-priority packets. (In the simulation, this does increase the average turnaround time particularly for bursty data, but it does decrease packet drop rate)

4. iSLIP, although it uses round-robin techniques, performs efficiently due to its virtual output queuing and ensures that even high-priority traffic gets faster service through smart scheduling mechanisms. (For complicated data, this has the best turnaround time as well as the lowest packet drop rate)

6.3 Which algorithm provides the highest fairness and how does the iSLIP algorithm improve performance over traditional round-robin methods?

1. Weighted Fair Queuing (WFQ) provides the highest fairness because it explicitly balances queue access based on predefined weights, ensuring that all input ports receive service proportionally to their priority.

2. iSLIP improves better performance over traditional Round Robin (RR) by implementing a three-phase request, grant, and accept mechanism. This allows iSLIP to avoid collisions and efficiently handle packets without the strict "one-after-the-other" approach of RR, leading to improved throughput and reduced packet drop rates in high-traffic situations.

6.4 Best scheduling algorithms for use in high-throughput router switch fabrics is: iSLIP

It offers low packet delay, highest throughput, lowest turnaround and waiting time, efficient handling of both high and low-priority traffic, and superior performance in bursty traffic scenarios. Its use of VOQ minimizes head-of-line blocking, leading to high throughput and fairness in resource allocation.

Appendix

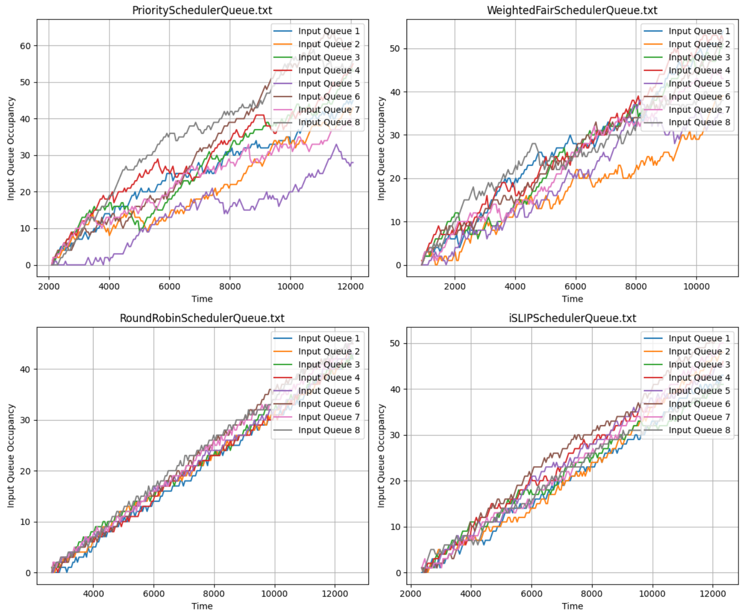

Figure 4: Input Queue Occupancy Graph for Uniform Traffic

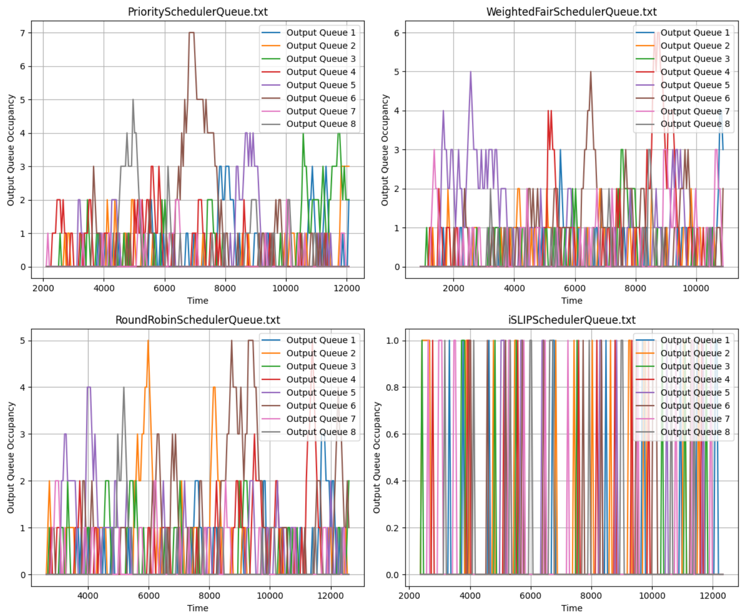

Figure 5: Output Queue Occupancy Graph for Uniform Traffic

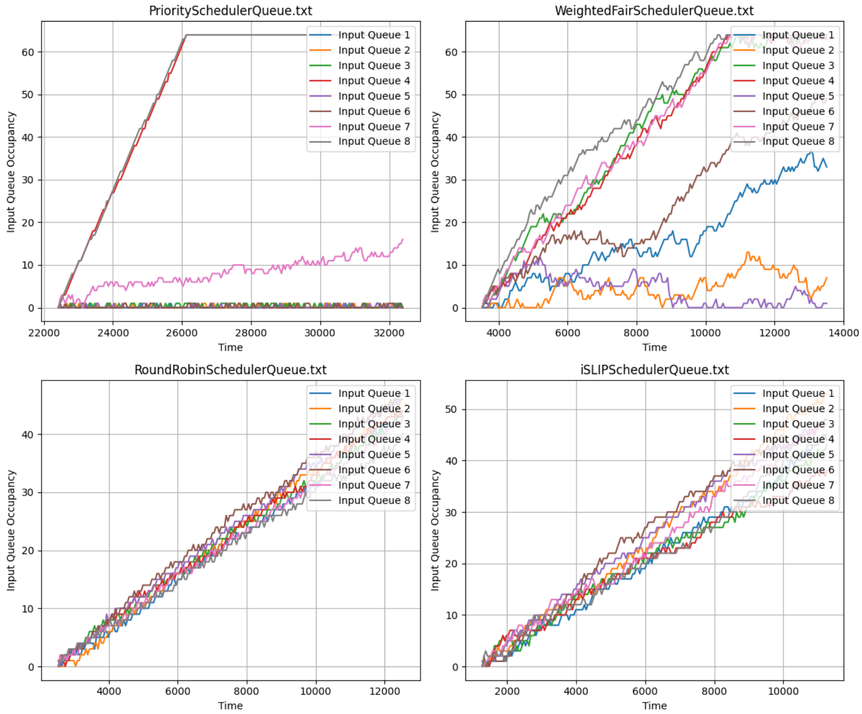

Figure 6: Input Queue Occupancy Graph for Non-Uniform Traffic

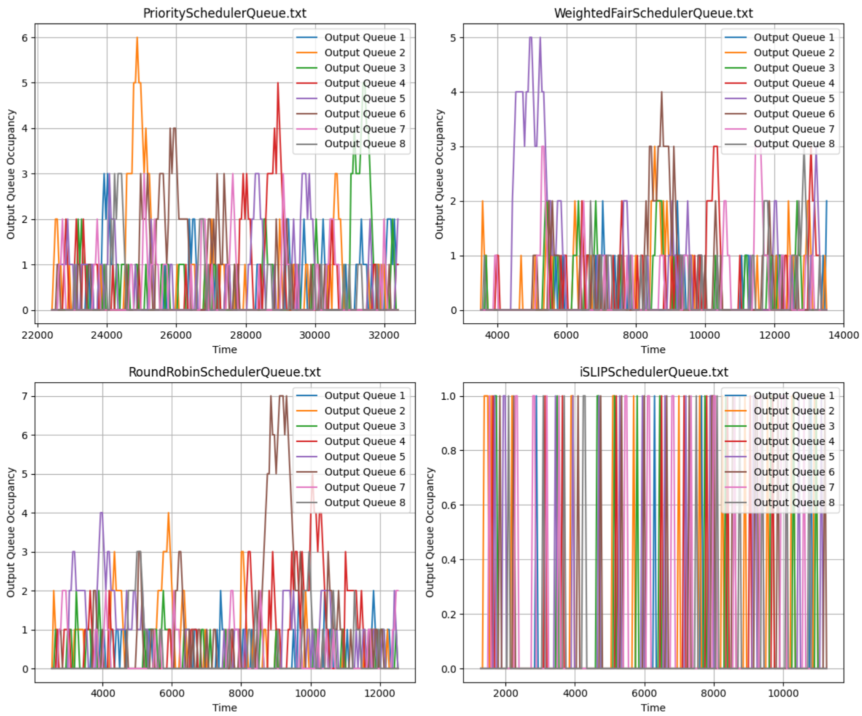

Figure 7: Output Queue Occupancy Graph for Non-Uniform Traffic

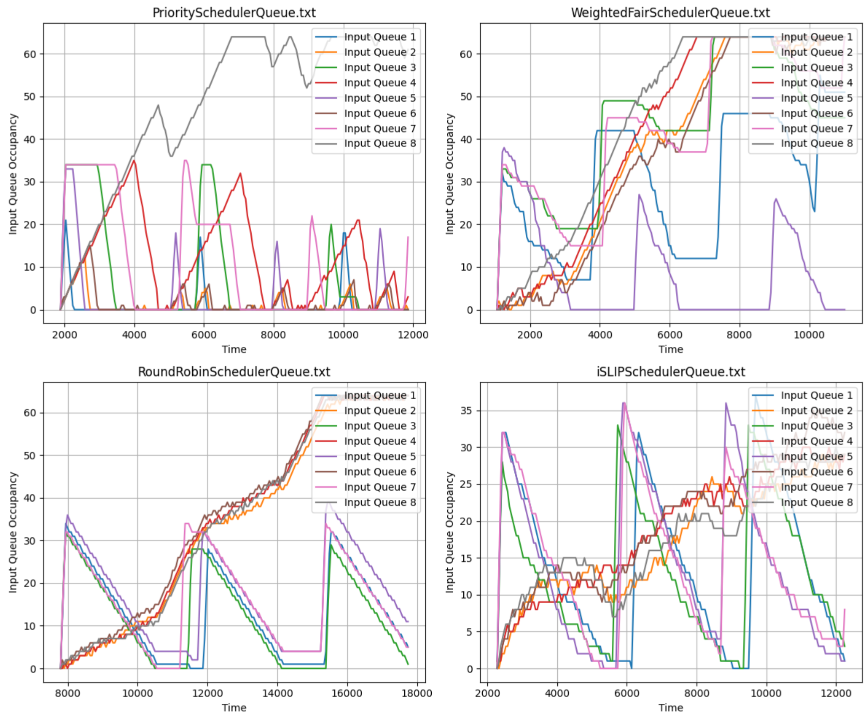

Figure 8: Input Queue Occupancy Graph for Bursty Traffic

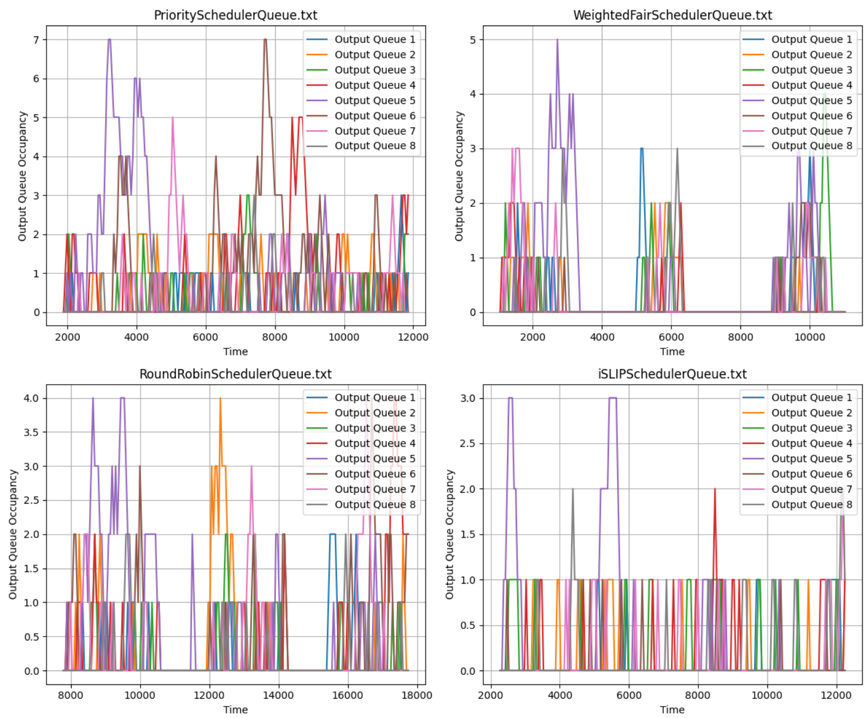

Figure 9: Output Queue Occupancy Graph for Bursty Traffic