Muhammad Aater Suleman
Youssef Hmamouche

Project 1
CS 372

PART II

BufSchedOutputStream.cc & .h

Implements the output stream to be used in part C. The main functions are the constructor and the write function. The write function, when called, checks for necessary conditions and then make threads update the buffer.

Also contains the consumer thread which reads from the buffer and outputs it onto the correctwrite.

BufStats.cc & .h
Contains the implementations of the state thread to be run with bsender.

InputStream.cc & .h
Standard file that was given to us

MaxNWScheduler.cc & .h
Contains the Implementation of the MaxScheduler. Important functions are WaitForTurn(); which will allow the schedules to be scheduled on the output.

NWScheduler.cc
Base file

OutputStream.cc
Base file

RecStats.cc
File to implement the receive stats thread.

Record.cc & h
Define an entry to the max queue

SFQNWScheduler.cc
The file implements the SFQ scheduler (part B)

ScheduledInputStream.cc
The file implements the scheduledinputstream for part a and b. The write function calles the scheduler.

ScheduledOutputStream.cc
The file implements the scheduledinputstream for part a and b. The read function calles the scheduler.

Stats.cc
base file

StatsTest.cc
base file

bsender.cc
The files implements the buffered sender. It uses Bstats instead of stats and BufScheduledOutputStream instead of ScheduledOutputStream

receiver.cc
contains the main read function. Include instanciation of the ReceiveStats and command line parser in main.

sender.cc
contains the main read function. Include command line parser in main.

PART II

The Network Scheduler

The goal of this assignment is to make sure the read and write operation on the socket is protected across threads. The goal is also to limit the rate of writing or reading to the sockets based on the network scheduler we are asked to implement. Before all, the different network schedulers enforce their policy through their algorithm.

Rather as a choice of familiarity, we have chosen to implement the "queue" as a vector. The available tools for these data structures make equally attractive.

The NWScheduler

NWScheduler is the scheduler that implements the locking and condition variable semantics.

This scheduler extends all the schedulers that we discuss below in detail.

The MaxNWScheduler

The core scheduler that we have implemented is based on the ticket example that Professor Vin explained in class. Every thread gets a ticket and waits for its turn. The ticket number is "distributed" to the threads based on first come first served (FCFS).

The scheduler iterates through the ready queue to find the thread with the smallest number. That is based on the FCFS.

As soon as the scheduler returns the ticket number, the thread holding that ticket number gets to write.

The Alarm function makes the thread sleep based on the rate it is assigned. It then goes and sends a broadcast to other threads that might want to run. Afterwards, the alarm function proceeds to exit.

The SFQNScheduler

This scheduler is based on the Start-time Fair Queuing algorithm that Professor Vin Engineered himself. The core scheduler proceeds and selects the thread with the smallest start time flag after iterating through the ready queue. Again, the alarm thread makes sure the thread currently running gets its CPU time before rescheduling.

This scheduling algorithm achieves fairness regardless of the load. The SFQ algorithm is well suited for interactive tasks that are running concurrently with other tasks.

Similar to the other schedulers, the SFQ scheduler is implemented in the WaitMyTurn function that is called right before the call to the higher level write function. The WaitMyTurn function implements multi-threading semantics to avoid multiple threads writing at the same time.

The Buffered Scheduler

The buffered scheduler resembles the producer/consumer scenario. This implementation is preferred to other methods because the writer is not put into a queue and delayed or kept
from writing.

We have created two threads: one is the consumer and the other is the producer. The producer is basically a thread writing to the buffer. The consumer is the thread that is actually taking data from the buffer to write to the socket. This mechanism is fairly simple in a sense that no thread gets put in the queue. The consumer comes and takes it from the Queue. They use two conditions to wait and signal, buffalo and bufEmpty signal.

We are not implementing a fair scheme for scheduling the threads that wait incase the buffer is full. We are relying on the condition variables synchronizing that for us. The project did not specifically give any instructions to implement an FCFS.

ScheduledOutStream/OutputStream

Depending on which scheduler implementation you are running, the "lower" level OutputStream write is governed by the "higher" level ScheduledOutputStream write. The difference among all these schedulers is the ritual WaitMyTurn function.

Right before the lower level write, we make the thread goes through the ritual WaitMyTurn to make sure when and if it has the right to proceed.

Every thread basically waits for its turn. The way we do it is that every thread that invokes the wait for my turn function will grab a ticket and start waiting on a condition variable.

When the thread that already had network gets back from sleep it wakes everyone by doing a broadcast and all the threads run but only the thread with the minimum ticket number gets to run.

ScheduledInputStream/InputStream

The relation between these two streams is similar to the relation between ScheduledOutStream and OutputStream. We make sure threads don't read unless they are allowed to through the multi-threading semantics.

MAXScheduler

The important function is wait for my turn called by the scheduled output stream. It maintains a FIFO and sleeping threads and wakes the next thread in the FIFO on its turn.

SFQScheduler Implementation

Every thread is assigned a start and a finish tag as soon as it comes in and calls the WaitMyTurn. The thread is then pushed on a vector. The stream flow ID is also updated with the new finish tag. Then the thread finds out what ticket the network is serving right now. If it is not its ticket it starts waiting on a condition. Once its its turn it will go ahead and update the global VirtualTime. Then it removes itself from the Queue and calls the alarm. Once alarm is done. It goes ahead and wakes threads sleeping on the network free. All the other threads then run and the one with the smallest start tag get to run.

5*i threads

The TA told us not to implement the 5 seconds wait since it will not be graded. We have not included the support in the software but here is how we had planned to do it:

```
struct blastarguments{
    OutputStream *os;
    int flowID;
}

typedef blastargument;
```

Inside main sender.cc:

for every flow that we create:

```
blastargument * arg= (blastargument *)malloc(sizeof(blastargument));

arg->os=  Whatever stream we are using at time

arg-flowID=ij;


sthread_create(thread, (void*(*)(void*))blast, (void *)arg);
```

Inside balst :

```
  os=arg->os;

  sthread_sleep(5*flowID,0);

  int got;
  int tot = 0;
  if(DEBUGGING){
    printf("Blast thread started\n");
  }
  while(1){
    got = os->write(request, BUF_SIZE);
    if(got <= 0){
      //   if(DEBUGGING){
             printf("Done sending after %d bytes; send thread done.\n", tot);
             // }
      return;
    }
    tot += got;
  }
}
```

PART VI

Testing strategies

MAXScheduler

We tested MAXscheduler starting from its base case i.e. only one flow sent from the sender. We temporarily modified the code to assign packet IDs to packets and saw if any packets were being lost. Only some trailing packets were being lost.

We analyzed the output data in MS Excel where we plotted both the cumulative and a second to second flow i.e. bytes that were transmitted that very second from that flow. The results looked promising and we are getting the correct bandwidth numbers incase of flows as many as 10.

SFQScheduler

We again started with a base class test with all flows having equal weights and then extended our tests to multiple flows with widely varying weights. The results were fairly promising.

Our attached graphs (non cumulative) depicts how the weights among the threads were basically shared every second. We did a statistical analyses of our data and the total number of packets received from each thread did obey the weights to a very good approximation. We again had some missing trailing packets which were able to trace by assigning IDs to those packets.

BufferedOutputStream

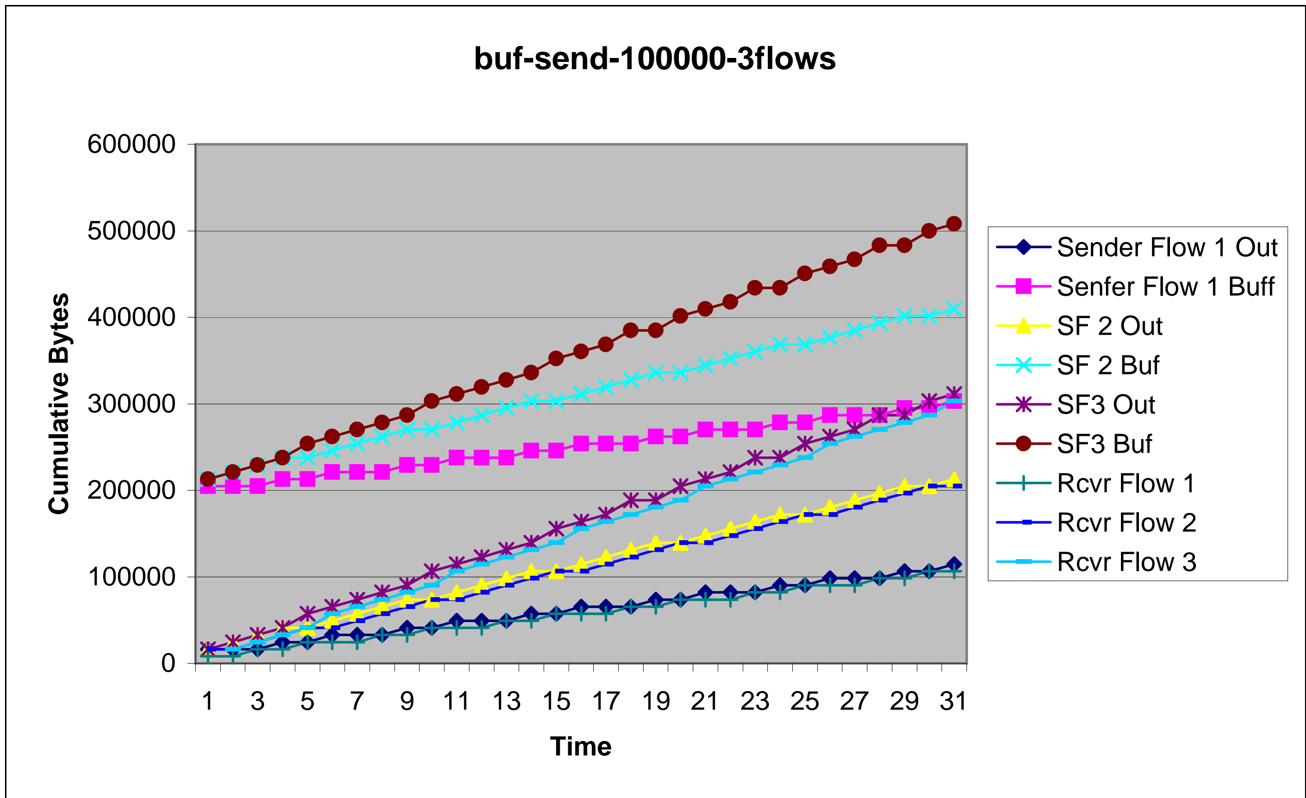
Testing buffered output stream was very similar to testing SFQ and MaxScheduler together. The first part resembled a MAX with unlimited rate and second was an SFQ.

We tried to test them individually to avoid variables in the system. So we put in debugging print statement to make sure that the threads write to the buffer in correct ordering.

After testing that write to buffer we started our consumer thread that would take data off the buffer. We used different packet sizes for consumer and produces to see and discrepancies but results resembled our intuition.

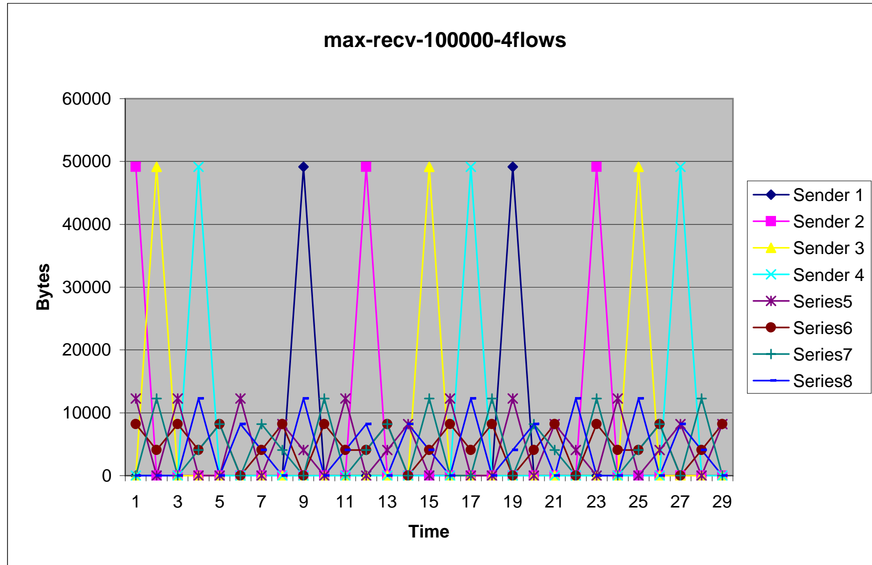
We analyzed the data and the graph is attached showing a cumulative flow of all the threads.

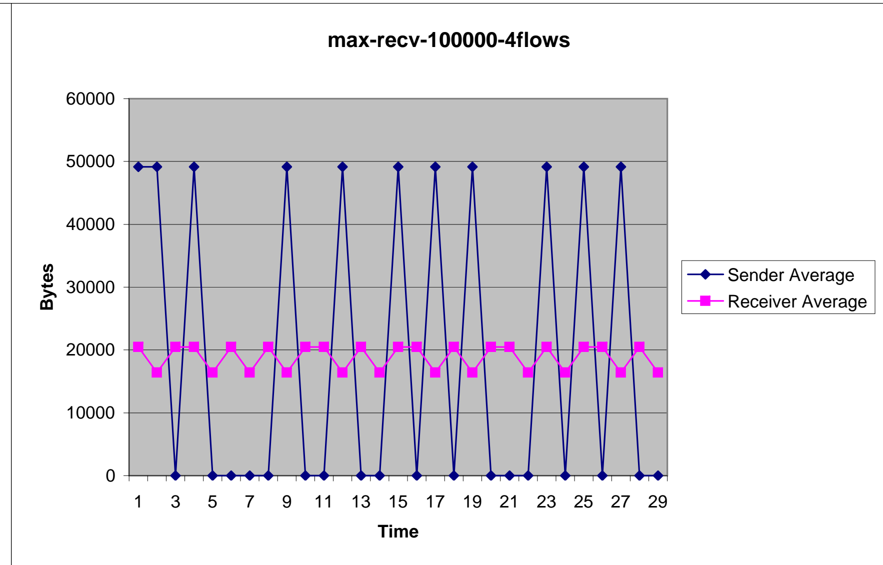
Send average 19660.8



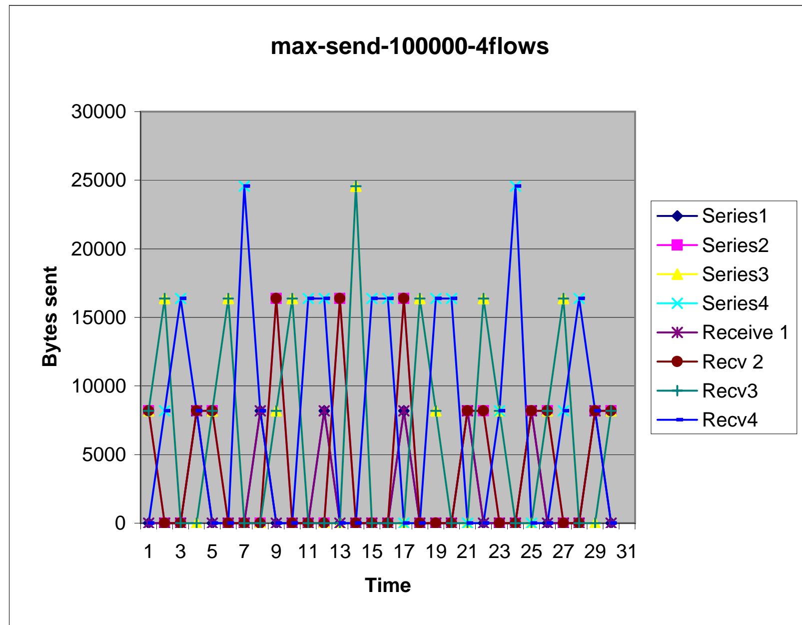
Send Average = 18643.86

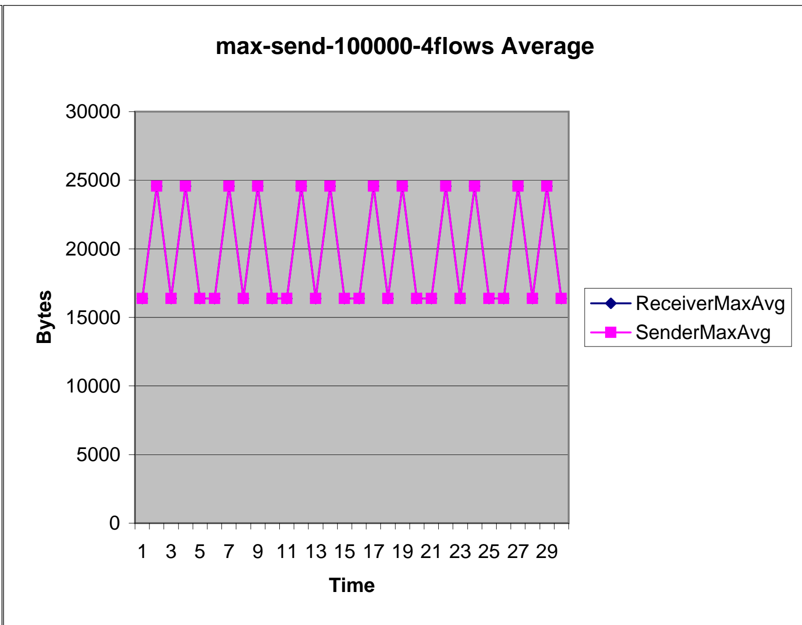
Receive Average= 18785.1





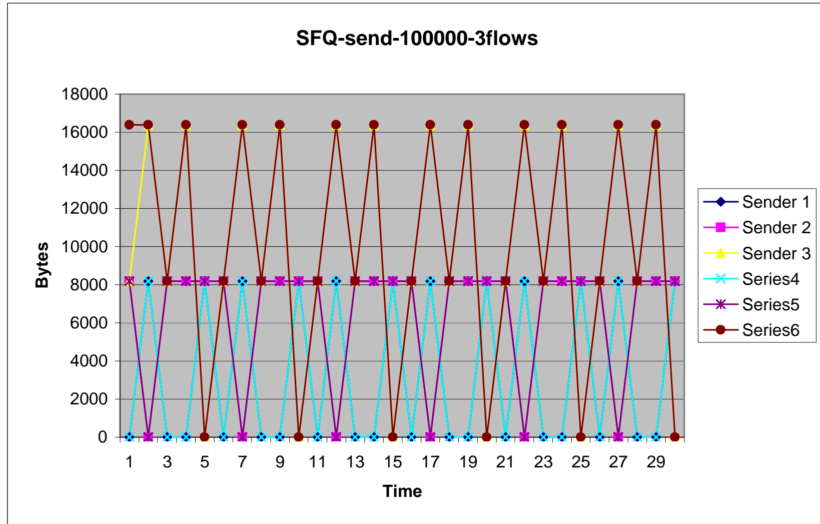
Average = 19660.8 Bytes/second

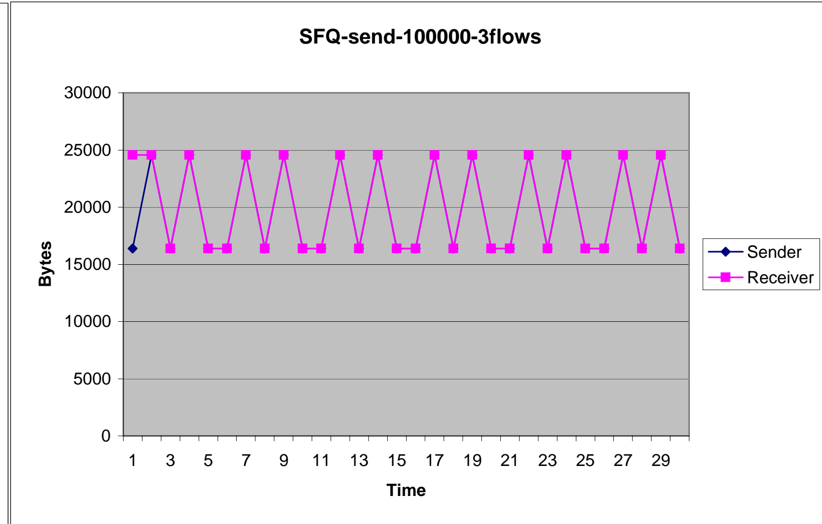


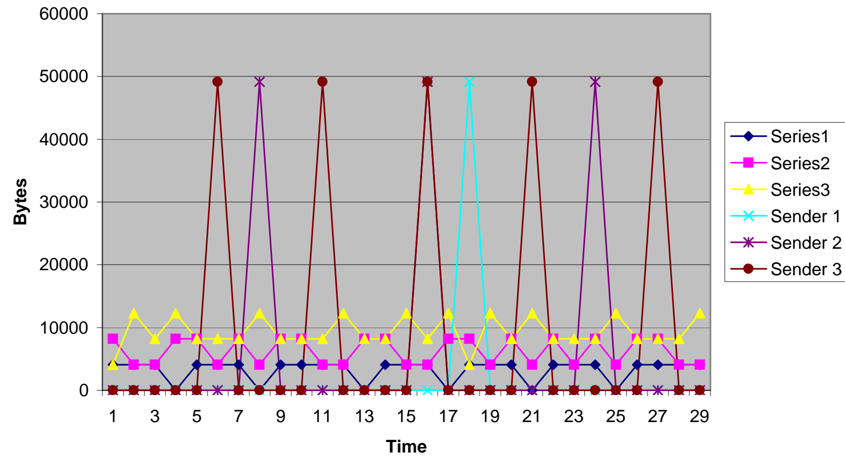


Average Send rate 19660.8

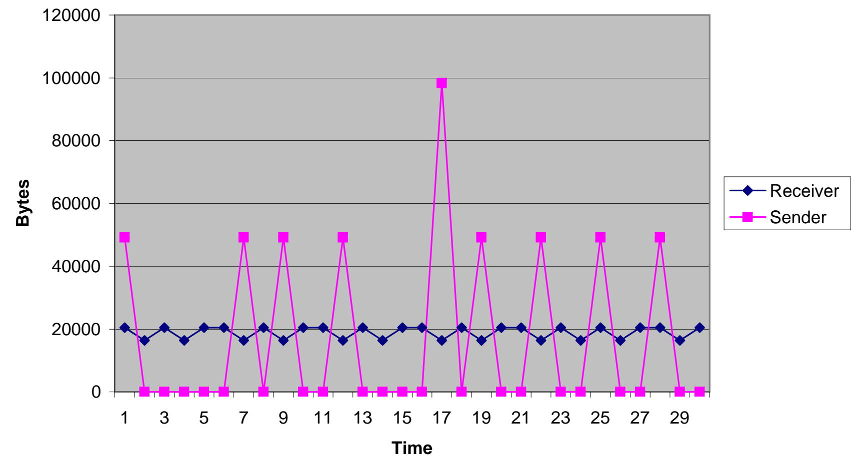
Average Recv Rate 19933.87







SFQ-recv-100000-3flows
Bytes
60000
50000
40000
30000
20000
10000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Time
Series1
Series2
Series3
Sender 1
Sender 2
Sender 3



SFQ-recv-100000-3flows Average
Bytes
120000
100000
80000
60000
40000
20000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Time
Receiver
Sender