

Concurrent Non-Blocking Priority Queue Implementation

CS 550 Advanced Operating System

Pradyot Mayank (A20405826)

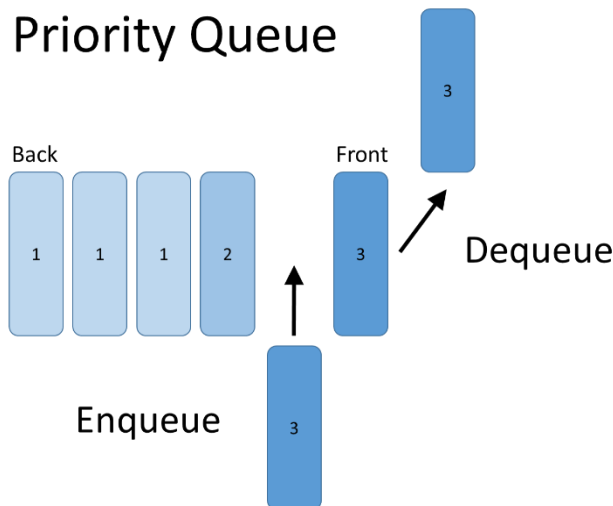
Nilesh Jorwar (A20405042)

Index

Contents	Page No
<i>Abstract</i>	<i>3</i>
<i>1. Introduction</i>	<i>3</i>
<i>2. Literature Review</i>	<i>5</i>
<i>3. Background Information</i>	<i>5</i>
<i>4. Problem Statement</i>	<i>8</i>
<i>5. Requirements</i>	<i>8</i>
<i>6. Design</i>	<i>9</i>
<i>7. System Architecture</i>	<i>10</i>
<i>8. Implementation</i>	<i>11</i>
<i>9. Output</i>	<i>11</i>
<i>10. Evaluation</i>	<i>14</i>
<i>11. Results</i>	<i>18</i>
<i>12. Tradeoffs</i>	<i>18</i>
<i>13. Improvements/Future Scope</i>	<i>18</i>
<i>14. Conclusion</i>	<i>19</i>
<i>15. References</i>	<i>20</i>

Abstract

We present an efficient and practical lock-free implementation of a concurrent priority queue that is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. Many algorithms for concurrent priority queues are based on mutual exclusion. However, mutual exclusion causes blocking which has several drawbacks and degrades the system's overall performance. Non-blocking algorithms avoid blocking, and are either lock-free or wait-free. Previously known non-blocking algorithms of priority queues did not perform well in practice because of their complexity, and they are often based on non-available atomic synchronization primitives. We have implemented our non-blocking concurrent priority queue in client-server architecture and a real-time extension of our work is also described. In our performance evaluation we compare our model with some of the most efficient implementations of priority queues known. The experimental results clearly show that our lock-free implementation outperforms the other lock-based implementations in all cases for 3 threads and more, both on fully concurrent as well as on pre-emptive systems.

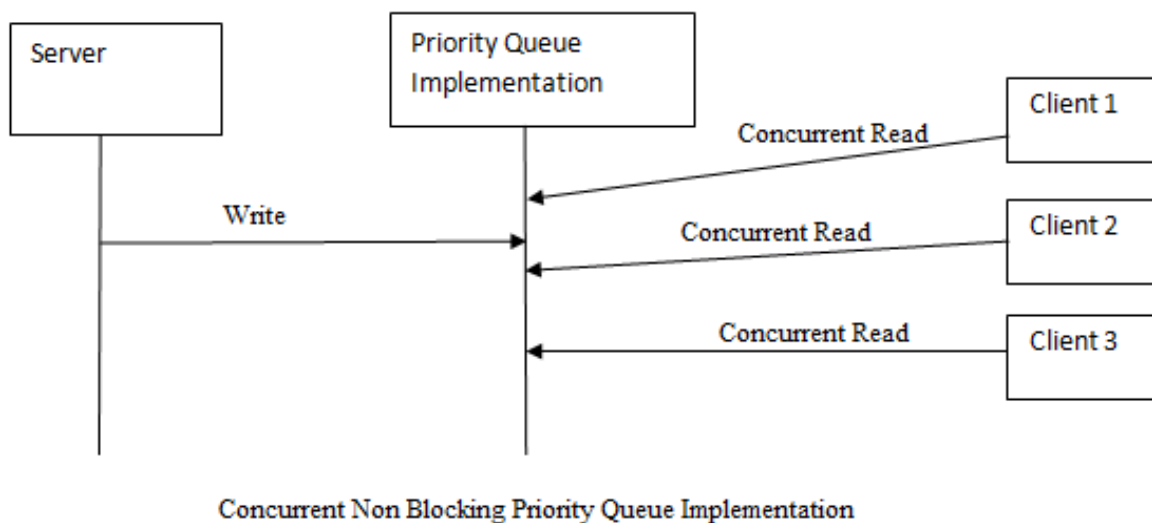


Enqueue and Dequeue operation in a Priority Queue

1. Introduction

By word “Non-Blocking”, we can infer that either it performs the operation requested by the thread or notifies the thread that it cannot perform the operation. Multithreaded environment is based on the concept of communication using various data structures like queue, maps, stacks

etc. We have to build the concurrent algorithm to facilitate the concurrent access to the data structure by multiple threads which makes the data structure a concurrent data structure. If the algorithm guards the concurrent data structure is non-blocking (no thread suspension), it is non-blocking concurrent data structure. Writing a thread-safe code is, at its core, about managing access to shared, mutable state. Making data structure thread-safe requires using synchronization to coordinate access to its mutable state; failing to do so could result in data corruption and other undesirable consequences. A priority queue is a data structure that is very often needed at places such as the process managers of operating systems, distributed event simulations, network bandwidth managers and sorting algorithms and thus may face severe contention. While priority queues are often implemented with heaps, they are conceptually distinct from heaps.



To ensure reliability of a shared data object in a concurrent environment, the most common method is to use mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance as it causes blocking, i.e. other concurrent operations cannot make any progress while the access to the shared resource is blocked by the lock. Using mutual exclusion can also cause deadlocks, priority inversion (which can be solved efficiently on uni-processors with the cost of more difficult analysis, although not as efficient on multiprocessor systems) and even starvation.

Blocking Vs. Non-Blocking

The primary difference between blocking and non-blocking algorithms is which threads are able to make progress in the code execution. In a blocking model, a single thread is allowed to prevent all other threads from making any progress. There is no guarantee that this single thread

is making progress itself or that it will ever stop blocking other threads. All locks are blocking, regardless of the implementation. A thread that has taken a lock can prevent others from continuing for an extended period of time for a variety of reasons. Some of these include being swapped out due to OS scheduling, experiencing a page fault, or terminating (e.g. due to a crash) without releasing the lock. On the other hand, a non-blocking algorithm guarantees that there exists a thread that is making progress. However, nothing is specified about which thread makes progress, meaning that starvation is still possible.

2. Literature Review

Queues are ubiquitous in parallel programs, but their performance is matter of major concern. There has been development in the past regarding thread safe, wait free algorithms on various data structures like Stack, Queue[1] by Hunt and Michael L Scott. Michael L Scott. And Maged Michael presented the non-blocking algorithm with one enqueue and one dequeue operations running concurrently using compare_and_swap and load_linked primitives [1]. Compare and Swap is based on the idea to compare the contents of memory location with the given value, if they are same, modifies the contents of that location to new value. But it was used for the purpose of achieving synchronization which would fail if memory location is updated by another thread in meantime in case of multithreaded environment. Lately, in 2004 again Michael Scott and William Scherrer came with implementation of dual data structures to describe the concurrent object implementation [2].

3. Background Information

With non-blocking and lock free implementation of priority queue we can achieve scalability and real-time response. Each process does not need to wait for its turn if there are multiple threads requesting the same resource. Unlike blocking concurrency algorithm, non-blocking concurrency algorithm either performs the operation or notifies the thread that operation cannot be performed at that time.

Lock Free Stacks

Stacks make for a simple case study, since the operations on stacks are very simple. The basic principle behind the implementation is that a thread will create a new version of the top of the stack and if no other thread has modified the stack, the change will be made public. However, now the focus turns to how this can be implemented. To accomplish this, a compare and swap (CAS) instruction is required so that the comparison and the write can be done atomically. This instruction is used to check that the top of the stack is not changed in between creating the new top and writing it to memory.

```

void push(Stack *s, Node *n) {
    while (1) {
        Node *old_top = s->top;
        n->next = old_top;
        if(compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

```

CAS is not guaranteed to succeed because another thread may perform its own CAS and change `s->top`. If this happens the CAS will see that `old_top` has changed and will try to perform the insert again with the new value. If the CAS succeeds it will push the new head onto the stack, and the push is done. To get reasonable stack behavior, we're implicitly assuming no starvation occurs.

The ABA Problem

Before discussing pop, let's reconsider what our CAS operation is actually checking for: that the top of the stack (the address of the Node, to be specific) is unchanged. This does not actually mean "the stack is unchanged." The ABA problem exposes a case where this isn't a strong enough check. The issue can be exposed with a sequence of push and pop operations.

1. Thread 0 begins a pop and sees "A" as the top, followed by "B".
2. Thread 1 begins and completes a pop, returning "A".
3. Thread 1 begins and completes a push of "D".
4. Thread 1 pushes "A" back onto the stack and completes.
5. Thread 0 sees that "A" is on top and returns "A", setting the new top to "B".
6. Node D is lost.

Pop

7. An implementation of pop that avoids the ABA problem can be implemented using a counter to keep track of the number of pops.

```

Node *pop(Stack *s) {
    while (1) {
        Node *top = s->top;

```

```

int pop_count = s->pop_count;
if (top == NULL)
    return NULL;

```

```

Node *new_top = top->next;
if (double_compare_and_swap(&s->top, top, new_top, &s->pop_count,
pop_count, pop_count + 1));
    return top;
}
}

```

Linked Lists

Implementing a lock-free linked list uses the same idea as stacks, but with the added complexity of operating on any position in the data structure.

Insert

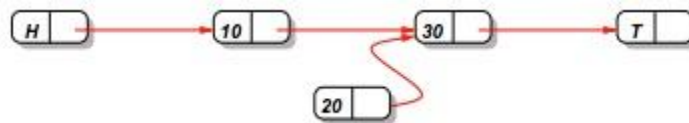
To implement insert, you first need to find the correct position in the list. Assume we have found this location and that we're inserting Node *n after Node *p. The insertion code is

```

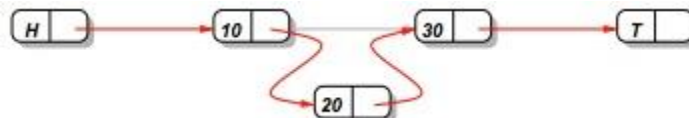
while (1) {
    n->next = p->next;
    Node *old_next = p->next;
    if (compare_and_swap(&p->next, old_next, n) == old_next)
        return;
}

```

The operation that we need CAS for is to update the next pointer for the node we're inserting after (*p) to ensure that another thread hasn't modified the same position. This process is shown below.



Before CAS, the node being inserted is already able to point to the following node. If CAS does not succeed, this is reassigned on the next iteration.



CAS is needed to change the next pointer of the node we're inserting after to ensure that no other thread makes a change at that position.

Deletion

As we saw with stacks, allowing for a second operation (pop) complicated the implementation. With linked lists, simultaneous insertion and deletion is not easy to account for. If we have "A" -> "B" -> "C", a problem case would be

1. Thread 0 begins to delete "B" from after "A".
2. Thread 1 begins to insert "D" after "B".
3. Thread 1 points "B" to "D".
4. Thread 0 points "A" to "C"
5. The delete operation removed "B" and "D".

To fix this, there must be a way to ensure that "A" is also unchanged when inserting after "B" and how to react when this happens.

4. Problem Statement

Concurrency in case of a priority queue is achieved using locks and thread blocking. When multiple threads try to add new element to a priority queue, only one of the threads gets access of the queue. Now while that thread is updating the queue, other threads are made to wait with the help of blocking. This effectively degrades productivity of the system plus it causes an additional overhead of thread communication and also increase the indefinite idle time for waiting threads and makes the Server slow due to incoming requests. Hence, we are implementing concurrent non-blocking priority queue to handle the emergencies as per priorities. This could be further extended to real world problems like in hospital management and producer consumer problems and in many market based problems.

5. Requirements

Software-

- JDK, NetBeans 8.0.2, Eclipse, JRE
- Linux/ Windows Operating System

Hardware-

- Single System for running the Server
- Multiple systems for running multiple Clients

- Virtual machines for Server and Clients

6. Design

Fig A and B shows Flow Chart and Use Case diagram of how our implementation works.

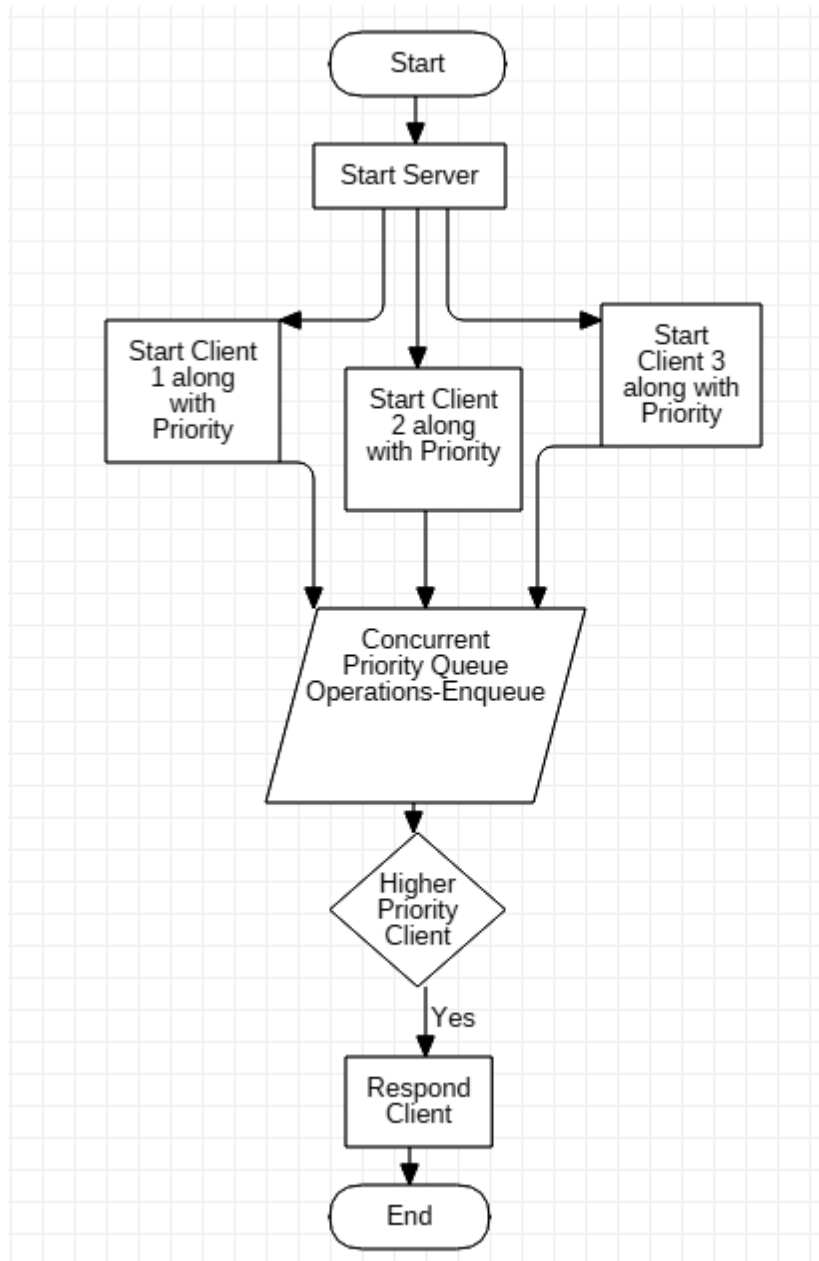


Fig. A. Flow Chart Diagram

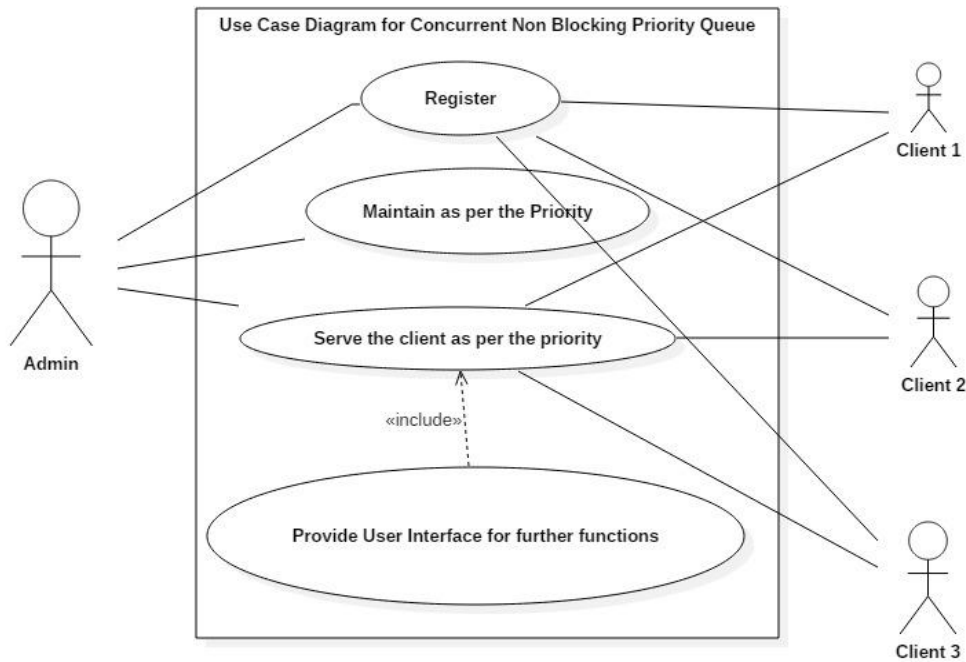
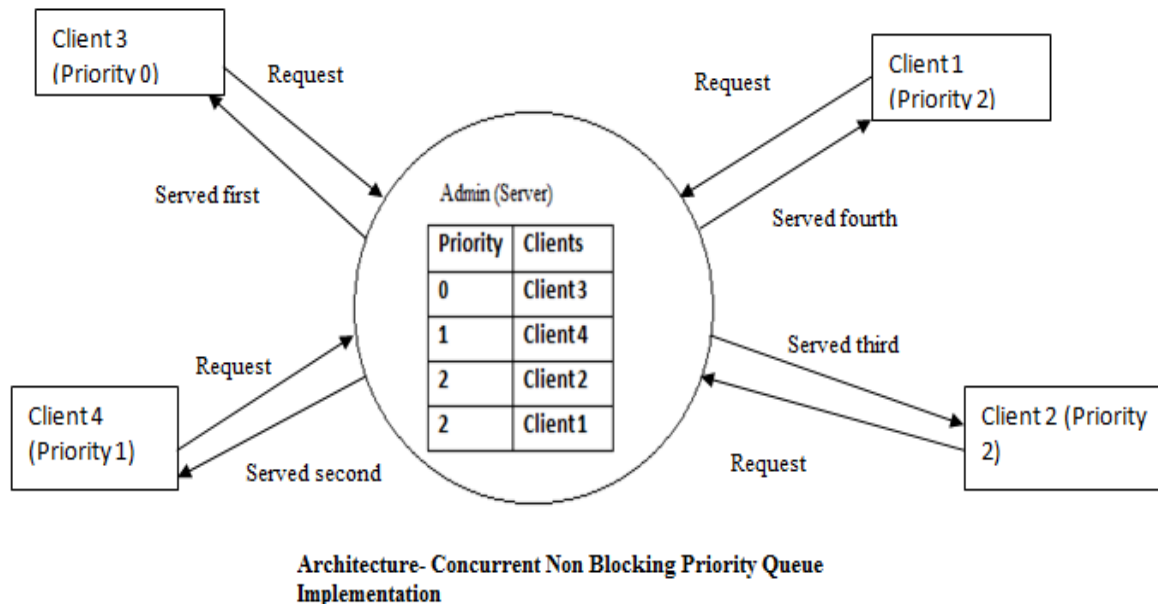


Fig. B. Use Case Diagram

7. System Architecture

In Concurrent non-blocking priority queue implementation, the Central server implements the priority queue for adding concurrent threads (clients) based on their priority. When concurrent threads/ clients connect with the server and requests for the server by messaging the server at the same time, the server which in turn waits for the concurrent requests from the clients and adds them in the queue as per the priority thus implementing Enqueue operation of the queue. While responding to the client based on the higher priority implements the Dequeue operation of the queue.



8. Implementation

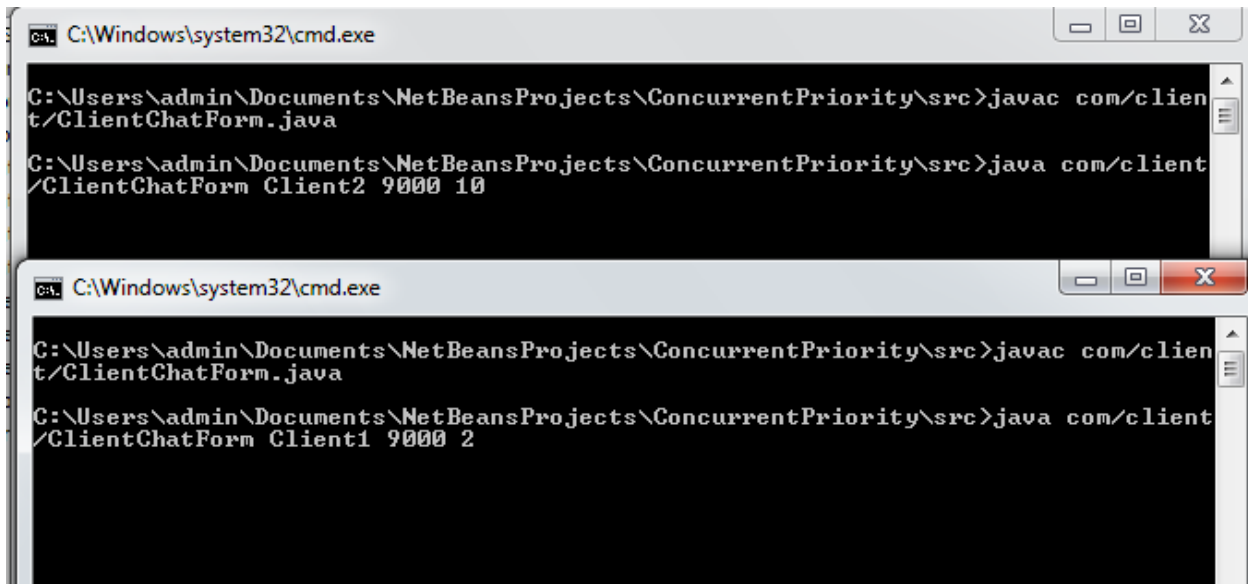
- Our implementation is distributed which has the server at remote location, waiting for the concurrent requests from the clients located on different machines which is implemented using the concept of the Multithreading.
- However, the Client Server interaction is implemented using the Socket Programming. This implementation also encourages the use of the Swing to provide the interface to the Server and Clients to interact with.
- Server has implemented its Enqueue and Dequeue operations in `conThreadStore(ArrayList<String> client, ArrayList<Integer>prio)` adding and removing the concurrent threads in the queue.

9. Output

Step 1: Enter the number of Concurrent Clients to run in “config.properties” file.

Step 2: Run the server from Command Prompt using batch File.

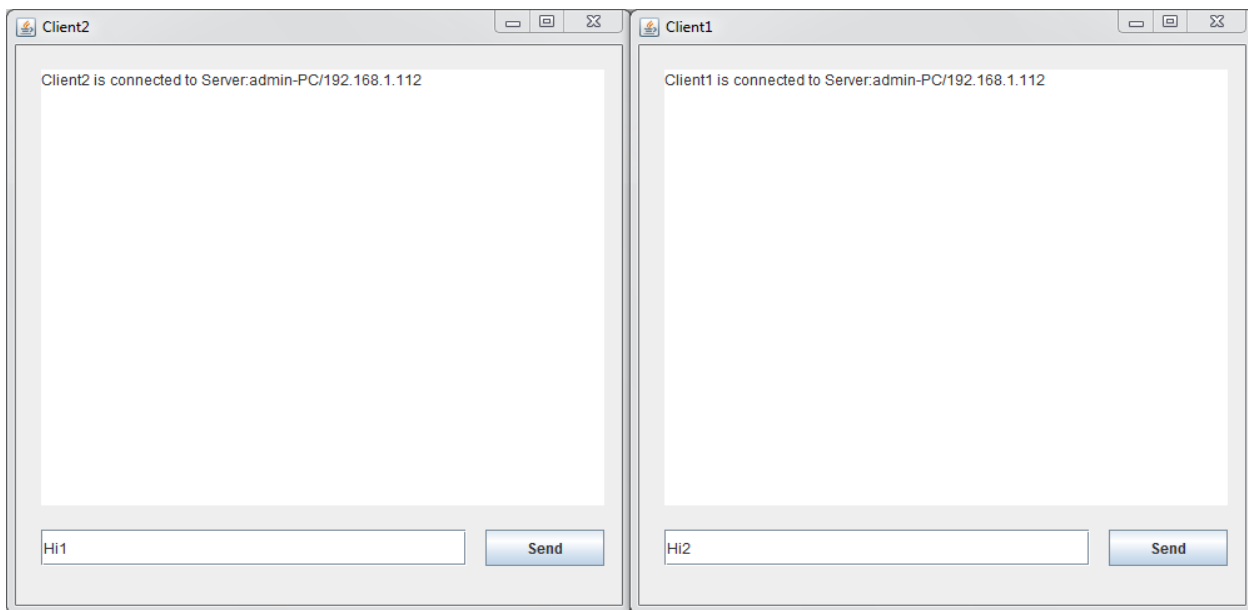
Step 3: Run the Concurrent Clients from same or multiple machines as follows.



```
C:\Windows\system32\cmd.exe
C:\Users\admin\Documents\NetBeansProjects\ConcurrentPriority\src>javac com/client/ClientChatForm.java
C:\Users\admin\Documents\NetBeansProjects\ConcurrentPriority\src>java com/client/ClientChatForm Client2 9000 10

C:\Windows\system32\cmd.exe
C:\Users\admin\Documents\NetBeansProjects\ConcurrentPriority\src>javac com/client/ClientChatForm.java
C:\Users\admin\Documents\NetBeansProjects\ConcurrentPriority\src>java com/client/ClientChatForm Client1 9000 2
```

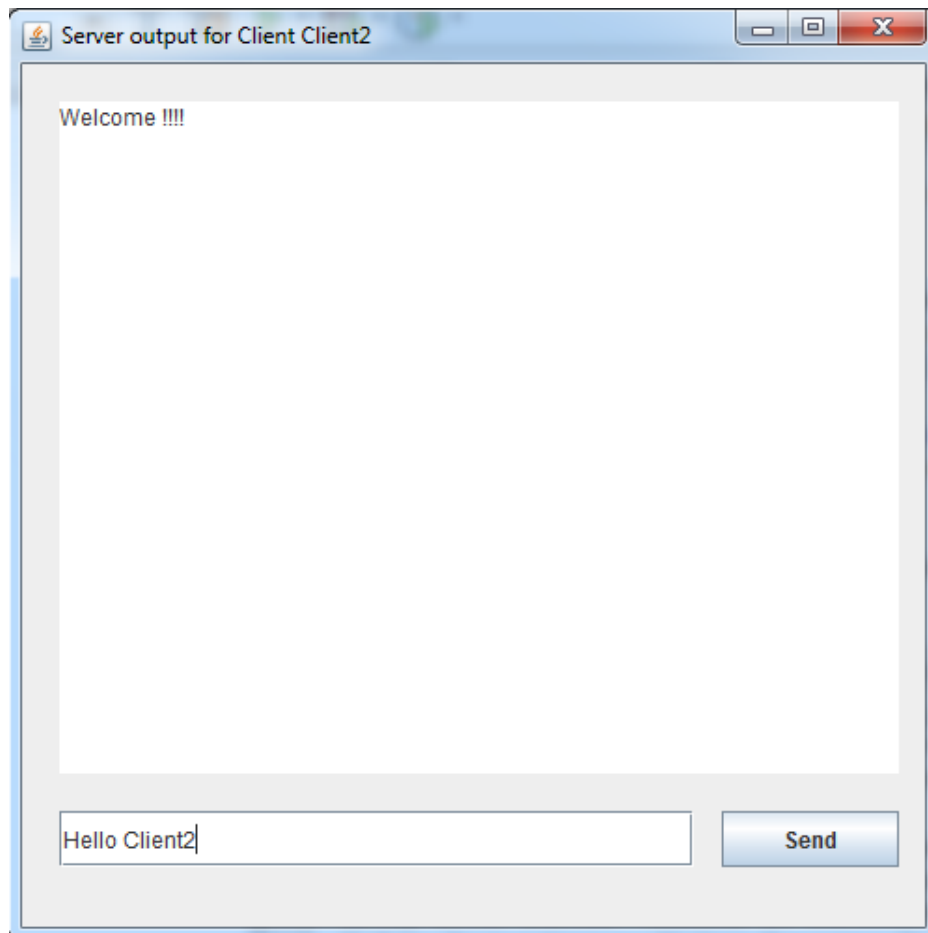
Step 3: Interact with the server by sending the message at the same time.



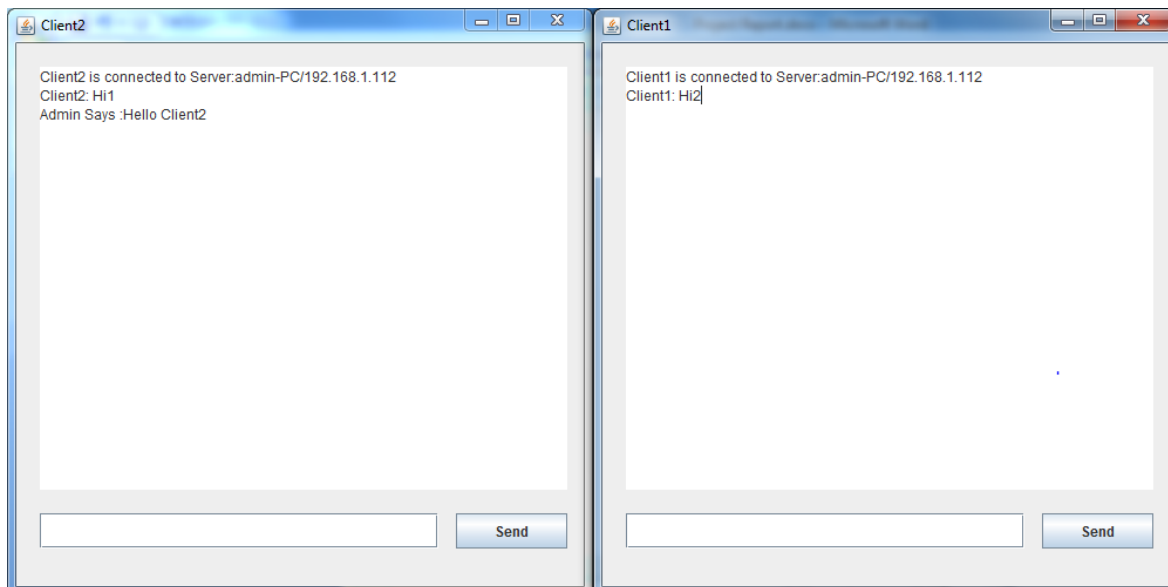
Step 4: Look for the server status for the sorted clients as per their priority.

```
Client Name==Client2 and its Priority==10
Client Name==Client1 and its Priority==2
Multivalue Map Size--2
Average time for enqueue operation in nanoseconds---1386674 for 2 users
Array after Sorted--Client1 2 Thread[Thread-1,5,main] Socket[addr=/192.168.1.112,port=49196,localport=9000]
Array after Sorted--Client2 10 Thread[Thread-0,5,main] Socket[addr=/192.168.1.112,port=49195,localport=9000]
Before Responding to Higher Priority Client
Average time for Dequeue operation in nanoseconds---71929556 for 2 users
```

Step 5: Send the response to higher priority Client from the Server.



Step 6: Check if the higher priority Clients gets the response from the Server.

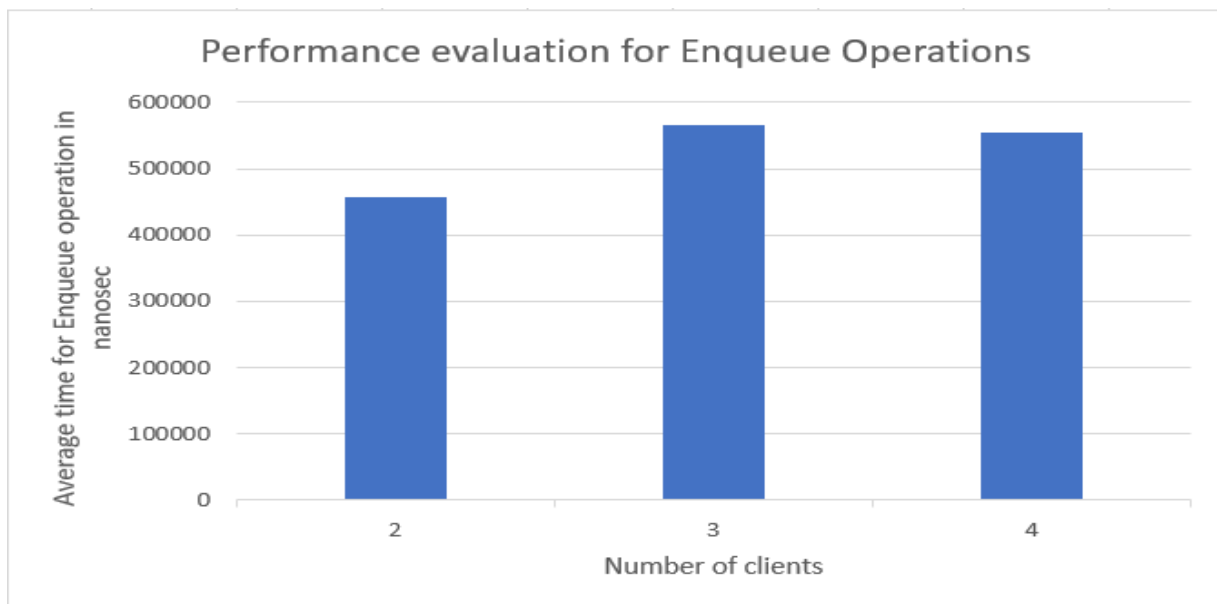


Step 7: Repeat above scenario from the step one for the given set of number of concurrent clients.

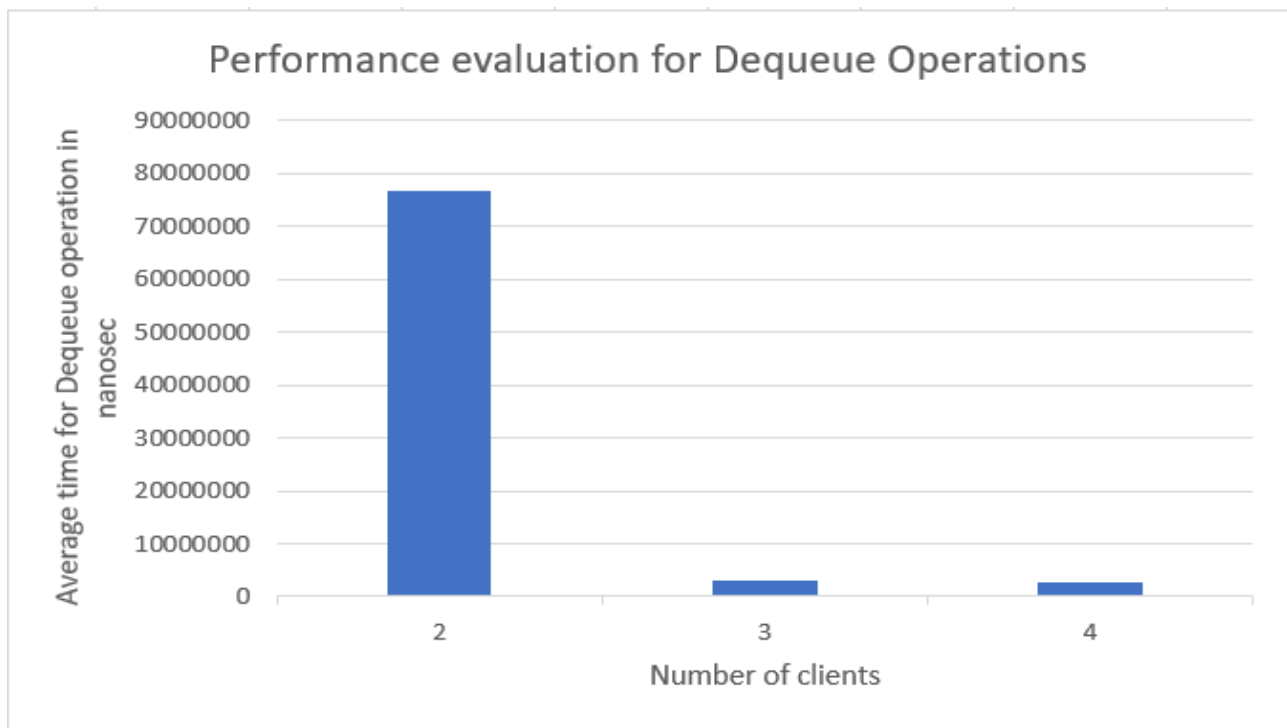
10. Evaluation

The evaluation of this project is comprised by the following:

Case 1: Average Response Time for Enqueue Operation at the Server Side for the no of Clients.



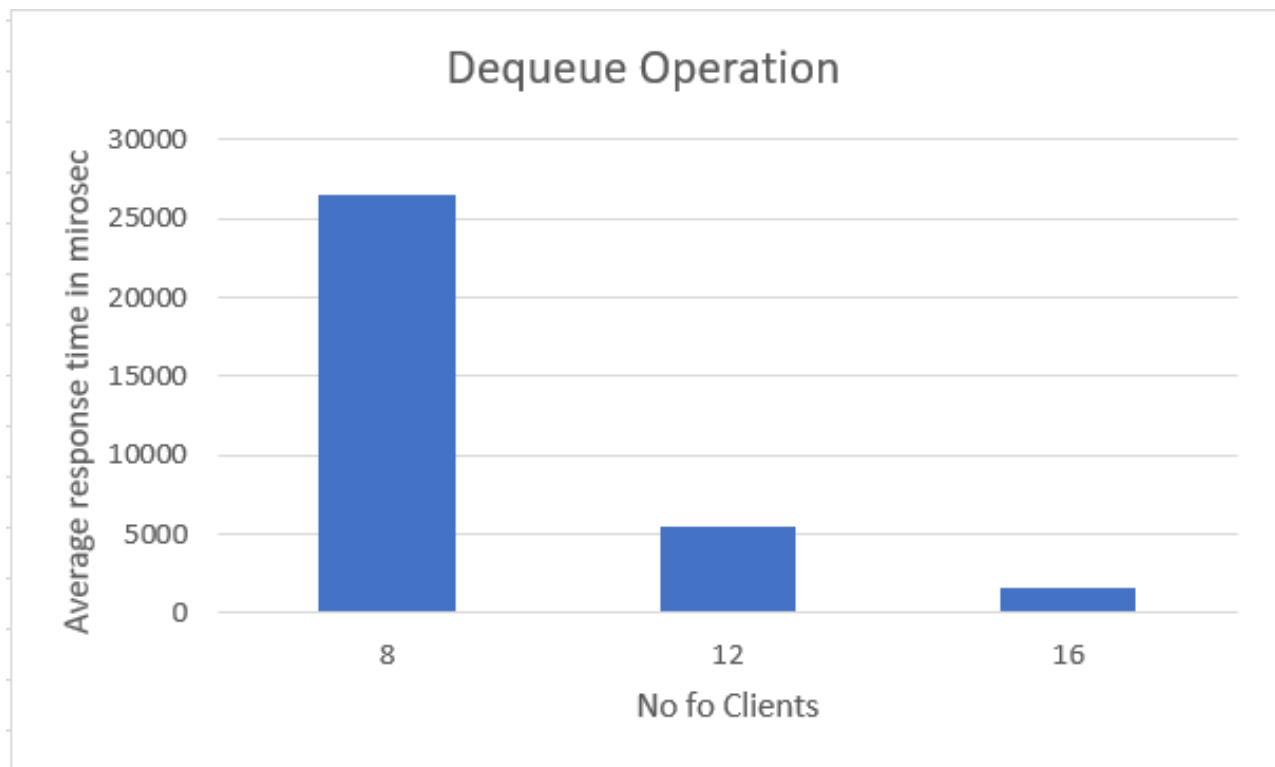
Case 2: Average Response Time for Dequeue Operation at the Server Side for the no of Clients.



Case 3: Average Response Time for Enqueue Operation at the server side for the number of clients.



Case 4: Average Response Time for Dequeue Operation at the Server Side for the no of Clients.



11. Results

```
Average time for enqueue operation in nanoseconds---455957 for 2 users
Array after Sorted--Client1 2 Thread[Thread-0,5,main] Socket[addr=/192.168.1.112,port=49724,localport=9000]
Array after Sorted--Client2 10 Thread[Thread-1,5,main] Socket[addr=/192.168.1.112,port=49725,localport=9000]
Average time for Dequeue operation in nanoseconds---76422438

Average time for enqueue operation in nanoseconds---565456 for 3 users
Array after Sorted--Client1 2 Thread[Thread-1,5,main] Socket[addr=/192.168.1.112,port=49727,localport=9000]
Array after Sorted--Client3 4 Thread[Thread-0,5,main] Socket[addr=/192.168.1.112,port=49726,localport=9000]
Array after Sorted--Client2 10 Thread[Thread-2,5,main] Socket[addr=/192.168.1.112,port=49728,localport=9000]
Average time for Dequeue operation in nanoseconds---2729755

Average time for enqueue operation in nanoseconds---553479 for 4 users
Array after Sorted--Client4 0 Thread[Thread-1,5,main] Socket[addr=/192.168.1.112,port=49735,localport=9000]
Array after Sorted--Client1 2 Thread[Thread-0,5,main] Socket[addr=/192.168.1.112,port=49734,localport=9000]
Array after Sorted--Client3 4 Thread[Thread-2,5,main] Socket[addr=/192.168.1.112,port=49736,localport=9000]
Array after Sorted--Client2 10 Thread[Thread-3,5,main] Socket[addr=/192.168.1.112,port=49737,localport=9000]
Average time for Dequeue operation in nanoseconds---2660463
```

12. Tradeoffs

Performance of the algorithm will be calculated based on complexities and big O notation.

- Used of ArrayList to maintain the priority and Client's name which adds the clients in $O(n \log n)$ time for the concurrent clients which will be reduced if the clients are not added constantly.
- Dequeue operation of the Priority Queue takes $O(1)$ time since the array is sorted and higher priority client is served.

13. Improvements/Future Scope

- Our solution would be used for the implementation of the Producer and Consumer problem with Priority.

- It can be efficiently used for the implementation of the Hospital Management where there is only one staff i.e. and will be serving higher priority patient with Axe on his head than those patients which are lined up in the queue.
- Scanning through a large collection of statistics to report the top N items - N busiest network connections, N most valuable customers, N largest disk users, these all problems could be resolved with the implementation of the concurrent non-blocking priority queue.

14. Conclusion

Our programming solution will implement non-blocking concurrency for priority queues. This priority based non-blocking data structure will work more efficiently for scheduling and sharing of the resources over the distributed network. We are going to check how high priority applications, resources get executed with minimal or no idle time. It is also going to check the performance improvement as compared to available non-blocking FIFO algorithms. This implementation could help solve lot of problems where the need of emergency exists. Variety of distributed applications where higher priority client need access would be served if implemented concurrent non block blocking priority queue.

15. References

- [1]MagedM. Michael MichaelL. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.
- [2]William N. Scherer III and Michael L. Scott. NonblockingConcurrent Data Structures with Condition Synchronization.
- [3]Carole Delporte, HuguesFauconnier,MichelRaynal. An Exercise in Concurrency: From Non-blocking Objects to Fair Objects ,2014
- [4]<http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#non-blocking-concurrent-data-structures>
- [5]<https://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html>
- [6]<http://blog.shealevy.com/2015/04/23/use-after-free-bug-in-maged-m-michael-and-michael-l-scotts-non-blocking-concurrent-queue-algorithm>
- [7]<https://secweb.cs.odu.edu/~zeil/cs361/web/website/Lectures/priorityQueues/pages/ar01s02.html>
- [8]<http://pages.cs.wisc.edu/~siff/CS367/Notes/pqueues.html>