# Wait-free Queue Algorithms for the Real-time Java Specification

Philippas Tsigas[1], Yi Zhang[2], Daniel Cederman[1], Tord Dellsén[1]

[1]Department of Computing Science
Chalmers University of Technology,
SE-412 60, Gothenburg,
Sweden

[2]School of Computer Science
University of Birmingham,
Birmingham, B15 2TT,
United Kingdom

## Abstract

*Efficient algorithmic implementations of wait-free queue classes in the Real-time Specification for Java are presented in this paper. The algorithms are designed to exploit the unidirectional nature of these queues and the priority-based scheduling in the specification. The proposed implementations support multiple real-time threads to access the queue in a wait-free manner and at the same time keep the "Write Once, Run Anywhere" principle of Java. Experiments show our implementations outperform the reference implementations, especially with high priority tasks.*

*In the implementations, we introduce a new solution to the "enabled late-write" problem discussed in [9]. The problem is caused by using only memory* read/write *operations. The new solution is more efficient, with respect to space complexity, compared to previous wait-free implementations, without losing in time complexity.*

## 1 Introduction

Using Java as the programming language for real-time and embedded systems has attracted certain academic and industry interests in recent years. In the States, the National Institute of Standard and Technology has organized the requirements working group for real-time extensions for the Java Platform. The Java Community Process Program has formed the Real-Time for Java Expert Group and produced the Real-time Specification for Java (RTSJ) [3, 4, 6]. Following the publication of the RTSJ, many papers have been published to address different issues in the specification [12, 8, 13].

To make Java more suitable for real-time programming, RTSJ enhances Java in several areas with better determinism and multithreading [3]. The enhancements include scheduling of real-time threads, memory management which allows applications to bypass garbage collection, introducing wait-free synchronization between real-time threads and non-real-time threads and others.

Among the above enhancements, we are interested in the wait-free synchronization between real-time threads and non-real-time threads. In this paper, we present efficient algorithmic implementations of wait-free queue classes defined in the RTSJ. Our implementations follow the "Write Once, Run Anywhere" principle which is one of the most important principles for Java and RTSJ and provide predictability with wait-free mechanism. In our implementations, we only use read/write operations which are supported by all Java runtime environments. At the same time, we provide formal proof for our implementation. We also compare the performance of our implementations with the reference implementations of RTSJ from Timesys.

The wait-free queue classes that are provided by RTSJ have been designed to enable communication between the real-time and the regular Java threads; they have a unidirectional nature with one side of the queue (read or write) for the real-time threads and the other one (write or read, respectively) for the non-real-time ones. The implementations presented in this paper are designed having the unidirectional nature of these queues in mind in order to gain efficiency and allowing multiple real-time threads to access the queue in a wait-free manner. To the best of our knowledge our implementations are the first unidirectional wait-free queue implementations that allows multiple real-time threads to access the queue in the literature.

The remainder of this paper is structured as follows. The next subsection describe related work. Section 2 provide a detail description of the problem. We present our implementations in section 3. The proof of correctness of our implementations is presented in section 4. We evaluate our implementations in section 5. Section 6 concludes the paper.

### 1.1 Related Work

Concurrent FIFO queue data structures are fundamental data structures used in many programs and algorithms and, as can be expected, many researchers have proposed implementations for them. Although there are many non-blocking implementations (see [11] for references), only few of them are wait-free. In a non-blocking algorithm, some operations are allowed to perform an unbounded number of steps when they are concurrent with other operations; this, of course, is not allowed in a wait-free algorithm. All previous constructions (wait-free or not) were targeted toward asynchronous systems; such constructions require hardware support for strong synchronization primitives such as Compare-and-Swap etc. These primitives are not available in the Real-Time Specification for Java. As a matter of fact in the RTSJ only read and write memory operations are supported. The reason is the hardware-independence property that the RTSJ wants to preserve.

Recent research at the University of North Carolina has shown that wait-free algorithms can be simplified considerably in real-time systems by exploiting the way that processes are scheduled for execution in such systems [1, 9]. In particular, if processes are scheduled by priority, then object calls by high-priority processes automatically appear to be atomic to lower-priority processes executing on the same processor. Consequently they show an implementation of Compare-and-Swap from reads and writes in a priority-based uniprocessor system [9]. In a consequent paper [2], a wait-free implementation of a linked-list from compare-and-swap for priority-based systems is presented. These results combined can offer an efficient implementation, with respect to time complexity, that satisfies the specifications of the wait-free queue classes in RTSJ. The space complexity of this implementation is $O(N * M)$ where $N$ and $M$ is the maximum number of concurrent tasks that the queue supports and the size of the queue respectively; the time complexity of this implementation is $O(N)$ for each task.

To enhance the concurrent programming ability of Java, the Java Community Process Program also worked out a specification for concurrency utilities, JSR-166 [], which is part of SUN Java JDK 5.0. In the concurrency utilities, an atomic primitive CompareAndSet, an equivalent of Compare-and-Swap, is provided. However, as stated in the specification, the hardware implementations of CompareAndSet may not be supported by some platforms; thus some form of internal locking may be used, and the method is not guaranteed to be non-blocking. The RTSJ introduces wait-free queues to avoid the dilemma introduced by locking which will be described later. Therefore, the atomic primitive CompareAndSet in the concurrent utilities cannot be used straightforward in implementing the wait-free queue required by RTSJ for real-time systems.

After the publication of RTSJ, a lot of research has been carried out to enhance and implement the ideas in RTSJ. For examples, in [12], the authors focus on asynchronous event handlers in RTSJ; paper [8, 13] focus on memory management issues in RTSJ. In the industry, Timesys provided the first reference implementation of RTSJ [10]. Recently, AICAS developed JamaicaVM which implements RTSJ. SUN also has a project named Mackinac to develop a commercial implementation of RTSJ. As Timesys is the first reference implementation of RTSJ it is relatively stable. In this paper, we use Timesys' implementation of RTSJ as the platform for evaluation and comparison.

## 2 Wait-free Synchronization in RTSJ

In this section, first, the wait-free synchronization mechanism and related features of RTSJ are presented. Then, we will discuss our understanding of wait-free synchronization in RTSJ.

The RTSJ is designed for multithreading priority-based uniprocessor systems. The application program must see the minimum 28 priorities as unique; for example, it must know that a thread with a lower priority will never execute if a thread with a higher priority is ready. If threads with the same priority are eligible to run, they will execute in FIFO order. The RTSJ provides wait-free queue classes to provide protected, non-blocking, shared access to objects accessed by both regular Java threads and `NoHeapRealtimeThreads` (`NHRT`). These classes are provided explicitly to enable communication between the real-time execution of `NHRT` and regular Java threads. Basically, there exist two different new queue classes in RTSJ: the `WaitFreeWriteQueue` class and the `WaitFreeReadQueue` class.

Both these queue classes are unidirectional. The information flow for the `WaitFreeWriteQueue` is from the real-time side to the non-real-time one, as shown in Figure 1. The information flow for the `WaitFreeReadQueue` is from the non-real-time side to the real-time one, as shown in Figure 2. When a `NHRT` wants to send data to a regular Java thread, it uses the `write` (real-time) operation of `WaitFreeWriteQueue` class. Regular threads use the `read` (non-real-time) operation of the same class to read information. The `write` side is non-blocking and wait-free, so that `NHRT` will not experience delays from the garbage collection. The `read` operation, on the other hand, is blocking. The `WaitFreeReadQueue` class, which is unidirectional from non-real-time to real-time, works in the converse manner.

The wait-free queue classes in RTSJ are used to solve a dilemma caused by `NHRT`, garbage collection and mechanisms for solving priority inversion in the specification. Garbage collection is an important language feature of

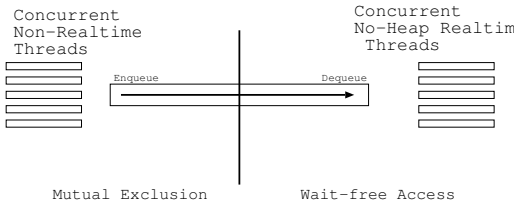


**Figure 1. The `WaitFreeReadQueue` class**

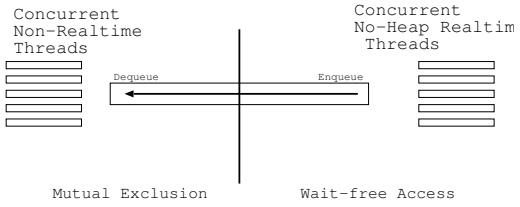


**Figure 2. The `WaitFreeWriteQueue` class**

Java and is kept in RTSJ. In RTSJ, regular Java threads and `RealtimeThread` cooperate with garbage collectors. The `NHRT` is introduced for threads which need to run without the intervention of garbage collection. When lock-based synchronization is used between threads, the priority inversion problem must be prevented by either priority ceiling emulation protocol or priority inheritance protocol, as required in the specification. Synchronization between `NHRT` and regular Java threads causes a dilemma if the synchronization is lock-based: regular Java threads may preempt `NHRT` to avoid priority inversion; in the mean time, garbage collection may preempt the regular Java threads and intervene in the execution of `NHRT`. For example, let us assume a NHRT $T_N$ shared information through a shared data object $SO$ with a regular Java thread $T_R$. The specification wants (a) the thread $T_N$ running without the interference of the garbage collection process and requires (b) that the thread $T_R$ cannot block the garbage collection process. The priority of $T_N$ is higher than that of thread $T_R$. To protect the consistency, the shared object is guarded by a monitor; to prevent the priority inversion problem, PCP or PIP is used at the same time. If $T_N$ preempts $T_R$ while it accesses the shared object $SO$, the priority of $T_R$ will be prompted to be higher than $T_N$. Now, if the garbage collection process starts, shall it preempt $T_R$? By the requirement (a), it cannot preempt $T_R$ which block $T_N$; if it preempted $T_R$, the action renders the introduction of `NHRT` meaningless. By requirement (b), it has to preempt $T_R$ to satisfy the consistency of JVM. To avoid the dilemma, RTSJ introduce wait-free synchronization between `NHRT` and regular Java threads.

In RTSJ, some requirements of wait-free synchronization are obscure. In the specification, it is said that "If two real-time threads intend to read from this queue they must provide their own synchronization." Should the synchronization between real-time threads be wait-free also? The reference implementation from Timesys uses lock-based synchronization between real-time threads. In this case, the wait-free queues are essentially single-writer/single-reader wait-free queues. As the RTSJ is published far ahead of the specification for concurrency utilities, JSR-166, it is unclear whether using CompareAndSet in JSR-166 complies with the specification.

To make things clear, we intend to develop wait-free queue implementations that satisfy the following requirements. First, if two or more real-time threads intend to operate on the same queue, they will cooperate with each other in a wait-free manner. This requirement will require multi-writer/multi-reader wait-free queue implementations. Second, we will not use CompareAndSet in JSR-166. Therefore, our implementations will keep the the "Write Once, Run Anywhere" principle and can be run in all environments which support RTSJ.

# 3 The Implementations of Wait-free Queues for RTSJ

The implementations of the two wait-free queue classes (`WaitFreeWriteQueue` and `WaitFreeReadQueue`) are quite similar algorithmically. In this paper, we present the implementation of the `WaitFreeWriteQueue` class to illustrate the ideas behind the constructions.

## 3.1 The Sequential Implementation

To simplify the presentation of our algorithm, we start with a simple sequential queue implementation. We will then discuss how to extend this sequential algorithm to a concurrent queue implementation with the specifications that we are looking for. The Java-pseudo-code for this sequential queue is shown in Figure 3.

As it can be seen in Figure 3 we implemented algorithmically the queue using a singly linked list. For efficiency reasons, we choose the front of the queue, where we only delete nodes, to be the head of the list, and the rear of the queue, where we only insert, to be the tail of the list. In this way we only use the operations of the linked list that modify the head and the tail of the list. In order to minimize the interference between the `write (enqueue)` and `read (dequeue)`[1] operations, we introduce a *dumbcell* in the

[1]Throughout the paper, the terms queue `write` and `enqueue` are used interchangeably. The same also holds for the terms queue `read` and `dequeue`. To distinguish between the queue `read/write` and normal read/write memory operation, we are using typewriter type style for the queue operations and serif type style for the memory ones.

```
1 public class SeqQueue {
    RTQueueCell head, tail;
3   RTQueueCell dumbcell;
    void SeqQueue() {
5     head = dumbcell;
      tail = dumbcell;
7     dumbcell.data = null;
      dumbcell.next = null;
9   }
    public java.lang.Object read() {
11    RTQueueCell temp;
      temp = (RTQueueCell)head.next;
13    if (temp != null)
        head = temp;
15    return temp;
    }
17  public boolean write(java.lang.Object object) {
      RTQueueCell temp;
19    temp = new RTQueueCell();
      temp.data = object;
21    temp.next = null;
      /*tail and tail,next will be shared
23      read/write in concurrent implementation*/
      tail.next = temp;
25    tail = temp;
      return TRUE;
27  }
  }
```

**Figure 3. The Sequential implementation of the queue**

empty list. In this way, when executing a `dequeue` operation, only the $head$ needs to be checked in order to see whether the queue is empty or not. If the $next$ field of the $head$ is $null$, the queue is empty. Therefore, the `dequeue` operation needs only to check the $head$ variable and only the $tail$ needs to be checked for the `enqueue` operation. For the initialization for the simple sequential queue we define a $dumbcell$, with `null` in its next field, and let the $head$ and the $tail$ of the queue point to it, statements 5 to 8 in Figure 3.

Figure 4 shows the structure of the cells of the linked list that we are using. The class `RTQueueCell` has two public members: one is for the data entry, the other is the next pointer that singly links the elements of the list.

### 3.2 The Concurrent Implementation

To extend the sequential version to a concurrent wait-free queue implementation, first we will use a simple announce-and-help scheme for the `enqueue` operations. The announce-and-help scheme uses the priority-based scheduler to achieve wait-freedom. This scheme is based on the task priorities to guarantee that an operation will finish in a bounded number of steps regardless of the status of the other operations, as follows: First, each enqueue-task announces the data (writes a pointer to the memory where the data are) that it wants to enqueue in a special $Announcement$ array. The enqueue-task with priority $i$ will use the $i$th position of the array. After the announcement step, the enqueue-task reads and helps the data that have been announced in the array one by one, starting from the lowest priority up to its own priority. During this helping phase, if an enqueue-task $A$ is not preempted by a higher priority task, then all current enqueue operations, with lower priority than the priority of $A$, that are announced will be helped/enqueued by $A$. If the enqueue task $A$ is preempted by a higher priority task $B$ during its helping phase, then there are two cases:

```
public class RTQueueCell {
        public java.lang.Object data = null;
        public java.lang.Object next = null;
 }
```

**Figure 4. Definition of the queue cell**

• $B$ is not an enqueue-task in the same queue: then task $A$ will continue its program steps after B finishes from the same queue-state from which it was pre-empted. Dequeue operations on the same queue are executed by tasks that have lower priority and therefore they can not preempt enqueue operations in the same queue.

• $B$ performs an enqueue operation on the same queue: in this case $B$ is going to announce its task and help all lower priority tasks that are announced and its own task that has just been announced. Therefore task $A$ will be helped by $B$. Because the priorities are bounded, there always exists a task which will not be preempted by another enqueue task. Therefore, all tasks that announced their operations will be helped (either by themselves or by higher priority tasks).

As stated before, we intend to only use read and write primitives in our implementation. Although reads and writes are very weak synchronization primitives in the context of general asynchronous systems, by exploiting the fact that the tasks are executed by priority, it has been shown that they are universal primitives for priority-based uniprocessor systems [9].

However, a problem named the "enabled late-write" problem in [9] arises from the use of memory read/write operations only. The "enabled late-write" problem arises when a low priority task $A$ is preempted while it is about to write to a memory position, and is preempted by other tasks that access and modify the same memory position. When task $A$ resumes running, it overwrites the previous "fresh" value with an "old" one. Anderson et al. [9] proposed a majority voting scheme to overcome the problem. Their scheme requires $2N - 1$ memory words to solve "the enabled late-write" problem for one word.

In this paper we propose a new more efficient scheme to face the "enabled late-write". The new scheme tries to avoid the problem from the beginning by:

1. Making sure that, when a task $A$ is preempted before writing to position $p$, all other tasks that write on $p$, (while $A$ is preempted) write the same value that $A$ wanted to write. In order to establish this, we guide the tasks to go through the same computational steps as $A$ when they have to decide about the value that they want to write on the same memory location.
2. When the above is possible, we organize the shared variables that might suffer from the "enabled late-write" problem as arrays that carry information that can be used algorithmically to determine the correct/new value of the variable.

We believe that, the same idea can be used when algorithmically designing other shared objects for the RTSJ.

```
public class WaitFreeWriteQueue
{
... ...
private MemoryArea MemPool;
private java.lang.Object[] Announcement;
private RTQueueCell[] tail;
private RTQueueCell head;
// get the minPriority from the scheduler
private int minPriority;
// get the maxPriority from the scheduler
private int maxPriority;
        ... ...
}
```

**Figure 5.** Shared private variables for `WaitFreeWriteQueue`

```
RTQueueCell dumbcell = new RTQueueCell();
... ...
Announcement = new
java.lang.Object[maxPriority + 1];
tail = new RTQueueCell[maxPriority + 1];

for (i=minPriority;i<=maxPriority;i++) {
Announcement[i] = null;
tail[i] = null;
}
tail[minPriority] = dumbcell;
head = dumbcell;
dumbcell.data = null;
dumbcell.next = null;
```

**Figure 6.** Initialization for `WaitFreeWriteQueue`

The wait-free part in this class is the part that implements the `enqueue` operations. The wait-free `write` operations also share the private variable $MemPool$ that hold references to a `MemoryArea`[2]. The shared private variables for our `WaitFreeWriteQueue` are as shown in Figure 5. All `RTQueueCell`s should be allocated from the `MemoryArea`. The $Announcement$ array is used to hold the different `enqueue` operations. The $tail$ and $Announcement$ arrays are of equal length, equal to the real-time priority level supported by the scheduler. For the head of the queue we use the simple variable $head$. $minPriority$ and $maxPriority$ are the minimum and maximum priorities that real-time threads can be assigned, respectively. This information can be obtained from the scheduler. All shared variables will be initialized when constructing the queue. The initialization is similar to that in the sequential version. Because we now use an array to represent the tail, we need to initialize this array in a way that makes it easy for the algorithm to find the correct `tail` (the $dumbcell$), when a task accesses the queue for the first time. When a task accesses the $tail$ array, it checks from the the cell of the lowest priority task to the highest to find a non-null cell. Henceforth, we initialize the cell corresponding to the lowest priority point to the $dumbcell$. During the initialization part, we also need to initialize the $Announcement$ array with the value `null`, which means that there are no announced operations. The pseudo-code for the initialization is shown in Figure 6. The initialization of the local variables is part of the pseudo-code description of the algorithm described in Figure 7.

Now, in order to extend the sequential version that we presented at the beginning of this section to the concurrent one that we are aiming for, we first need to make sure that the shared read/write operations to the $tail$ and the $tail.next$ variables (the shared variables of our implementation where overwriting might take place) do not suffer from the "enabled late-write" problem. The wait-free `enqueue` operation is presented in the `write` function below. The announce-and-help scheme, that is used in our implementation, uses the priority-based scheduler to achieve wait-freedom. Each priority is mapped to the respective entry of the array $Announcement$. An `enqueue` operation first gets the priority of its thread, then it allocates a free cell from the memory area assigned to the queue. The memory area is where the queue and its internal elements are allocated. After writing the data in the free cell, the task announces this cell in the $Announcement$ array at the index that is associated to its priority. This constitutes the last part of the announcement phase. This is, as we will see later, the "linearization point" of the enqueue operation at the linearizability history. After it announces the object that it wants to enqueue in the $Announcement$ array, a task will enter the helping phase that was described at the be-

[2]The RTSJ introduces the $memoryarea$ concept, which is a region of memory outside the garbage-collected heap that you can use to allocate objects. The RTSJ uses the abstract class `MemoryArea` for this.

ginning of this subsection. The helping phase is described in relation to the implementation pseudo-code in Figure 7. During the helping phase, an `enqueue` operation with priority $i$ helps the tasks with priority $j \leq i$ that have been announced in the array $Announcement$, one at a time, starting from the operation with the smallest priority that it can find (statement 18 in the implementation). For each such operation, it finds the tail of the queue (statements 23-33 on the protocol); then puts the data announced at the end of the $tail$ of the queue; then changes the $tail$ variable to point to the new position; and finally cleans $Announcement[j]$.

The wait-free queue classes we designed are used to provide communication between NHRTs and regular Java threads. To untangle the effect of garbage collection, statical memory management is needed for nodes of the queue class. Statical memory management is not the subject of this paper; but, a simple scheme is presented in [5]. Other better and more efficient schemes are possible. In [5], we describe implementations of the other methods supported by the `WaitFreeWriteQueue` class.

```
public boolean write(java.lang.Object object) {
2 boolean find = false;   int i,j, mypriority;
  RTQueueCell tempcell, temptail=null;
4 java.lang.Object tempAnnounce;
  java.lang.Thread currentone;
6 //Find current task's priority
  currentone = java.lang.Thread.currentThread();
8 mypriority = currentone.getPriority();
  //Allocate a cell in the MemoryArea
10 try {
    tempcell = MemPool.newInstance(RTQueueCell);
12  tempcell.data = object; tempcell.next = null;
  } catch(OutofMemoryError x) { return false; }
14 //Announce current task's operation
  Announcement[mypriority]=tempcell;
16 /*Enter helping phase and help tasks with
    lower priorities and itself*/
18 for(i=minPriority;i<=mypriority;i++) {
    tempAnnounce=Announcement[i];
20  if (tempAnnounce == null) continue;
    //Try to find the actual tail
22  find = false;
    for(j=minPriority;j<=maxPriority;j++) {
24  if (tail[j]!=null)
     if (tail[j].next == null) {
26     find = true;
       break;
28   }
    }
30  //Continued in Figure 8
```

**Figure 7. Wait-free enqueue operation for the `WaitFreeWriteQueue`**

Figure 9 shows the lock-based `read` operation of the `WaitFreeWriteQueue` class. It's a straightforward implementation that uses mutual exclusion to serialize concurrent dequeue operations.

```
    if (find)
32  //No preemption. The actual tail is found.
      temptail =  tail[j];
34  else
    //Preemption detected! There are 2 possibilities
36    if(Announcement[i]!=null) {
    /*Low priority tasks are preempted and helped but
38    the help is not completely. Help the task with
      priority i to update tail and Announcement array*/
40      tail[i]=(RTQueueCell)tempAnnounce;
        Announcement[i] = null;
42      continue;
      }
44    else
    /*Current task is preempted and helped by a higher
46    priority task that helped all low priority tasks.*/
        return true;
48
    /*Check whether current task preempt a lower priority
50    task when it was on statements 64 and 65?*/
    if(temptail==tempAnnounce)
52  {
    //If so, help it to update the Announcement
54    Announcement[i] = null;
      continue;
56  }
    /*Check whether current task is preempted by a
58    higher priority task that has helped current
      task to complete its operation?*/
60  if(Announcement[i]==null)
      return true;
62  //Enqueue the announcement
    temptail.next=tempAnnounce;
64  tail[i]=(RTQueueCell)tempAnnounce;
    Announcement[i] = null;
66 }
  return true;
68 }
```

**Figure 8. Wait-free enqueue operation for the `WaitFreeWriteQueue` (continued)**

## 4 Correctness Proof

In the helping phase two sets of variables are used, the $tail$ array (tasks help to enqueue data at the tail of the queue) and the $Announcement$ array; both are shared variables and can be read and written by different tasks. In the implementation, the value of a variable $Announcement[i]$ changes from $null$ to a non-null value when a task with priority $i$ announces its `enqueue` operation. The value of the same variable changes back to $null$ when the item that the `enqueue` operation wanted to enqueue was enqueued by the same operation or by another higher priority enqueue operation. If there are $e$ enabled writes that are ready to write to $Announcement[i]$ then at least $e - 1$ of them are

```
 public synchronized java.lang.Object read() {
2RTQueueCell tempcell;
 tempcell = head.next;
4if (tempcell != null)
 head = (RTQueueCell) tempcell;
6return tempcell;
 }
```

**Figure 9. Lock-based dequeue operation for `WaitFreeWriteQueue`**

helping operations and have priority higher than $i$ and want to change the value of the $Announcement[i]$ from *non-null* to $null$. The one "enabled late-write", that might exist, is the write with priority $i$ that wants to announce a new enqueue operation. This write will not be scheduled before the other pending writes, with higher priority, take place, and thus, its write will not be overwritten by them. The above proves the following lemma:

**Lemma 1** *$\forall i$, $minPriority \leq i \leq maxPriority$, $Announcement[i]$ will not suffer from the "enabled late-write" problem.*

**Lemma 2** *When a task $A$ is preempted just before it writes to the $tail$ array, a higher priority task will write the same content to the same position in the $tail$ array.*

**Proof:** The decision of what to write to the tail is based on the contents of the $Announcement$ and $tail$ arrays. If a higher priority task preempted task $A$ just before $A$ was to write the $tail$ array, then, since, nothing changed on the $Announcement$ and $tail$ of the object from the time that $A$ read them, the higher priority task, that preempted A, will compute the same value to write to the $tail$ array. □

If we would have used a simple $tail$ variable for the queue, as it is used in the sequential implementation of the queue, the "enabled late-write" problem could have happened in the $tail$ variable. To solve that problem, we organize the $tail$ of the queue as an array. Each location in the array corresponds to the respective priority. All tasks with the same priority will be executed in a FIFO order and use the same location in the array. Each enqueued item from a task with priority $i$ will become the tail of the queue once, and the $i$th index of the $tail$ will point to it. In our construction, all tasks that try to help a task with priority $i$ that has been announced, are going to write to the $tail$ array at the index that corresponds to priority $i$. For example, when a task $A$ is helping with task $C$, it is preempted by another high priority task $B$ and has an enabled write on the $tail$array. Then the new task $B$ will help the same task $C$ also and will go through the same computational steps and will update the same entry of the $tail$ array with the same value as the preempted enabled write of task $A$. This is guaranteed from Lemma 2. In this way, the "enabled late-write" problem can not take place in any $tail[i]$ variable. This sketches a proof of the following lemma.

**Lemma 3** *$\forall i, minPriority \leq i \leq maxPriority$, $tail[i]$ will not suffer from the "enabled late-write" problem.*

Now, we need to give a way for the tasks to read the $tail$ array and compute the real tail of the queue. Each item in the $tail$ array has been the real tail of the queue at some point in time but only one of them is the current tail of the queue. In our implementation, there is at most one $tail$ entry that has the value $null$ on its next field. As we are designing a concurrent queue, an enqueue operation can be preempted anywhere; a task $A$ can be preempted between statement 63 and 64 by a task $B$. The $tail$ array then will have no element with the value $null$ on its next field. The actual tail in this case should point to the object enqueued by a task $C$, which is being helped by task $A$ ($A$ executes statement 63 and 64 i only when it is helping another task). During the helping phase of its enqueue operation, task $B$ need to find the tail of the queue and uses the local variable *temptail* to store it. In the pseudo-code, when task $B$ executes statements 23-29, it goes through the $tail$ array from the lowest priority to the highest priority and tries to find the one index in the array with $null$ in the 'next' field, if there is one. If there is no overlapping with enqueue operations, task $B$ will find the index with $null$ in the next field. It will store the value in $temptail$ (statement 33).

**Lemma 4** *$temptail.next$ will not suffer from the "enabled late-write" problem.*

**Proof:** When a task $A$ with priority $j$ helps a task with priority $i$ that has been announced, where $i < j$, all items in the announcement array from $minPriority$ to $i-1$ should have the value $null$ because task $A$ starts its helping phase from $minPriority$, and, tasks with priority less than $j$ can not preempt task $A$ and make changes in the announcement array. Before task $A$ updates the next field of the tail of the queue (statement 63), nothing changes in the tail array and the announcement array. If a task $B$ with priority $k$, where $k > j$, preempted task $A$, task $B$ will add its own announcement in position $k$ and nothing between $minPriority$ to $j$ in the announcement array will change. Therefore task $B$ will help the announcement of the task with priority $i$ and will find the same tail and make the same decision with task $A$ and finally put the same value as $A$ on $temptail.next$. □

If task $B$ overlapped with other `enqueue` tasks, then task $B$ might not find an index on the array with $null$ in the next field. If this happens, task $B$ has already enough information to find the actual tail of the queue and help the preempted task to update the tail of the queue. To see this,

let us look at the possible ways that the above could have happened; there are two cases:

• Task $B$ preempts the lower priority task $A$, when $A$ was between statements 63 and 64; e.g. $A$ had just finished enqueueing the data before updating the tail of the queue. The actual tail of the queue at this point is the task which is being helped by both task $A$ and task $B$. Task $B$ will help task $C$ to update the tail when $B$ runs statements 40-41.

• Task $B$ is preempted by a higher priority task $D$ and $D$ updates the $tail$ array in such a way that task $B$ misses the actual tail of the queue when $B$ is scheduled back. In this case, $D$ will help all lower priority tasks. So, task $B$ just needs to stop its helping phase and return. $B$ will detect that it has been helped and return in statement 47.

The above sketches a proof that items are going to be put on the singly link-list one after the other.

Since different tasks are going to try to help the same task, we need to show that an item is not going to be enqueued more than one time. That is the reason that statement 51 is used from task $B$ to detect that it has preempted a task $A$ when $A$ was between statements 64 and 65 of its pseudo-code. When the preemption happens, the announcement has been added to the queue as a tail but not been cleaned, which has been read by task $B$ in $temptail$. If such a preemption is detected, the task $B$ will help task $A$ to clean the announcement array, when task $B$ executes statement 54. As both of them want to write $null$ at the same position, no "enabled late-write" problem exist. Statement 60-61 is used from task $B$ to detect that it had been preempted by a higher priority tasks $D$ and to conclude that task $D$ has helped the task that B was helping when preempted.

The following lemma also proves that it is necessary and sufficient for a task to help other tasks with priority up to its own priority.

**Lemma 5** *When a task $A$ with priority $i$ announces an enqueue data in the $Announcement$ array, all elements of the array from $i+1$ to $maxPriority$ have the value $null$.*

**Proof:** Assume toward a contradiction that $Announcement[j]$ is not $null$, where $j > i$. Then there must exist a task $B$ with priority $j$ that announced its enqueue object in $Announcement$ array and the announcement by task $B$ hasn't been "cleaned". $Announcement[j]$ is cleaned as the last step of the enqueue operation. The task $A$ must preempt task $B$ to announce its enqueue object in $Announcement$, in order to preempt task $B$, $i > j$ must hold. This is a contradiction,

As the contents of the $Announcement$ array from index $i+1$ to index $maxPriority$ are $null$ when task $A$ announce its operation, there is no need to help them. It is sufficient to help tasks with priority up to $i$. As task $A$ can preempt any lower priority task after it has announced, it is necessary to help them. □

The access of the queue is modeled by a history $h$. A history $h$ is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, a `write` operation or a `read` operation. An operation is called complete if there is a response event in the same history $h$; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation $q$ "occupies" a time interval $[s_q, f_q]$ on a linear time axis ($s_q < f_q$); we can think of $s_q$ and $f_q$ as the starting and finishing time instants of $q$. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in a history denoted by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that $q_1$ ends before $q_2$ starts; Operations incomparable under $<_h$ are called *overlapping*. A complete history $h$ is linearizable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_{\mathrm{h}}$ that respects the specification of the object [7].

In the remains of this section, we prove that our implementation is a concurrent linearizable queue implementation. In order to do so, we will show that any possible history ($<_h$), produced by our implementation, can be extended to a total order ($\rightarrow_{\mathrm{h}}$) by using a "linearization point" for each operation. The "linearization point" of an operation is an atomic point on its execution, during which the operation takes effect.

**Lemma 6** *The* write *to the announcement array is the "linearization point" for the* `write` *operations.*

**Proof:** By Lemma 5, when a task $A$ with priority $i$ executes statement 15, all items of the announcement array from $i+1$ to $maxPriority$ have the value $null$. Task $A$ will help all operations announced in $Announcement$ from the lowest to its own priority. Enqueue operations with lower priority than $i$ that have been announced by executing statement 15, will be enqueued before $A$'s announcement on the $anouncement$ array. If the current task $A$ is preempted by a higher priority task after executing statement 15, the announcement will be enqueued before the announcement of the task with higher priority. So, the execution order of statement 15 in the `write` operation extends the precedence partial order to a total order that respects the FIFO specifications of the `WaitFreeWriteQueue` class. □

**Lemma 7** *The* read *of the $next$ field of the $head$ of the queue is the "linearization point" for the* `read`*s of the queue.*

**Proof:** Since, mutual exclusion is used between `read` operations on the queue, the order in which they get access to

the critical section totally orders them. But as the read operations of the queue have lower priority than all the write operations of the queue, they can be preempted and run concurrently with write operations. As all high priority tasks will appear atomic to a low priority task, a write operation will only be observed if it starts executing before statement 3 of the read operation. By selecting then the execution of the statement 3 of a read operation as its "linearizability point", all operations are totally ordered with a relation that extends the precedence relation and respects the specification of the WaitFreeWriteQueue class. □

The lemma below proves that our queue implementation is a FIFO one and that no enqueued element gets lost. For simplicity we introduce write($empty$) operations in the history when the queue is empty.

**Lemma 8** *In a complete history such that* write($x$) $\rightarrow_h$ write($y$)*, then* read($x$) $\rightarrow_h$ read($y$)*.*

**Proof:** ¿From the assumption, we have that write($x$) $\rightarrow_{\mathrm{h}}$ write($y$) which means $x$ is announced before $y$. If there is no overlapping, $x$ will be put in the list before $y$ as in the sequential version. If overlapping exists, by lemma 5, the task $A$ who announces $y$ has higher priority than the task who announces $x$. As task $A$ will help from the task with lowest priority to itself, it will put $x$ in the list before $y$. As read uses mutual exclusion, only one read operation processes the list from the head to the tail. So read operations will find $x$ first. □

The lemma below proves that dequeue operations dequeue items that have really been enqueued.

**Lemma 9** *In a complete history, if $x$ is* read*, then it has been* writed*, and* write($x$) $\rightarrow_h$ read($x$)

**Proof:** The linearizability point of the read($x$) is the point where the read operation reads the $next$ field of the $head$. Because a write operation announces its operation and the announcement takes place before the helping phase, and in the helping phase the announcement will be put in the next field of a $tail[i]$. If $x$ is read, then some task must write in the field during its helping phase. Helping an announcement can only happen after it has been announced by some task in the announcement array. So, the $x$ read by a task must have been written before the read operation. □

The above lemmas give us the following theorem.

**Theorem 1** *Our algorithm for the* WaitFreeWriteQueue *is a linearizable FIFO concurrent queue without the "enabled late-write" problem.*

## 5 Experimental Evaluation

In this section, we present the experimental results of evaluating our implementations. We implemented WaitFreeWriteQueue and test it under the RTSJ reference implementation from Timesys. We compare the proposed implementation with the one in the reference implementation from TimeSys and the one from [2].

For the TimeSys implementation, we don't have the source code. To our knowledge, their implementation only has support for wait-free access from one thread at a time. For example, if two real-time threads would like to do wait-free writes to a queue at the same time, lock-based synchronization needs to be applied between the two threads. Therefore, when several real-time threads want to access the wait-free implementation from TimeSys, we have to choose some kinds of protocol to avoid the priority inversion problem. In our experiments, we choose the priority ceiling emulation protocol: whenever a real-time thread want to access the queue, we will raise the priority of the thread to the highest priority and restore the priority afterward.

All experiments were run in the following environment. We used the reference implementation for RTSJ from TimeSys. The version of the JVM from TimeSys used was 1.0.0 build 547. Since the reference implementation was based on Linux, we installed it on a Linux distribution from TimeSys. The distribution we used is version 4.1 build 155 which is optimized for standard Pentiums with kernel 2.4.7. The computer used for the experiments is a dedicated workstation with a 1100 MHz Intel Celeron processor with 128kb L2 cache and 512MB memory (with 100 MHz clock frequency). To get good timing information, we used the Pentium specific instruction rdtsc (read time stamp counter) to get runtime cycle information from hardware performance counters built into Pentium processors.

Due to the space limitation, we only report the results about the memory requirements of the three implementations and worst-case response times of the proposed algorithm and the reference implementation from Timesys. Extensive experiments have been carried out. Readers who are interested in the results are referred to [5].

Figure 10 shows the results of memory consumption of the three implementations with different queue sizes. In the experiments, we created a queue in immortal memory and measured the amount of free space before and after. From the figure, we can see the reference implementation from Timesys has the smallest memory consumption. However, this implementation only support a single real-time thread to do wait-free operation on the queue. If several real-time threads want to access the queue, lock-based synchronization and protocols to prevent priority inversions must be in place. The memory consumption of the lock-based synchronization and related protocols is in the JVM and Linux kernel. Measuring such memory consumption is complex and beyond the scope of this paper. The implementation of the proposed algorithms requires about 20 times more memory than the reference implementation at application

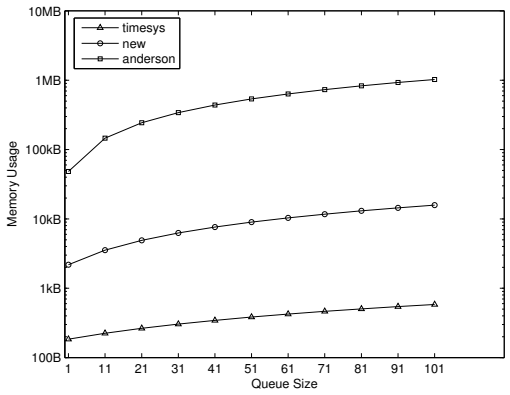


**Figure 10. Memory Consumption of Different Implementations**

level. The implementation from [2] uses more memory than the others, because their implementation of Compare-and-Swap requires a lot of memory.

The experiments to measure the response time overhead were carried out with 2 to 16 real-time threads. In the experiments, the lowest priority thread only performs read operations on the queue and all other threads only perform write operations. To simulate a real-time execution environment, we use the following parameters to keep the processor utilization to $90\%$ for all experiments. In the experiment with $n$ threads, we name the threads from 0 to $n-1$. Each thread runs a periodic task. Thread 0 is the lowest priority thread and thread $n-1$ is the highest priority thread. Let the computation time of the highest priority task be $c$. The computation times and periods of all threads can be determined as following to keep the processor utilization to $90\%$.

$$T_i = (n-i) * 20 * C, \quad i = 0, ..., (n-1)$$
$$C_i = \begin{cases} (n-i) * C, & i = 1, ..., (n-1) \\ (19-n) * n * C, & i = 0 \end{cases}$$

In the above formulas, $2 \leq n \leq 16$; $T_i$ is the period of task $i$; $C_i$ is the computation time of task $i$.

We use the three `WaitFreeWriteQueue` implementations in the above task sets. We measure the response time overheads of tasks for different implementations. The response time overhead for task $i$ is defined as $T_{ro} = t_f - t_s - C_i$ where $t_s$ is the time impact of release jitter will be the same for that the system starts to run the job and $t_f$ is the time that the job is finished. We choose to measure the times at the application level. We assume the overhead introduce by JVM will be the same for experiments with the same number of threads. The response time overheads of write tasks with the lowest and middle and highest priorities are presented in figure 11 for the implementation of the proposed algorithm and the reference implementation. The response times reported are the worst response times in $50$ periods for each cases. As one can see from the figure, the difference between the two implementations is small for the lowest priority write task. For the lowest priority task, the computation time of the task is much larger than the time spend on operations on the queue. Therefore, the performance different between the two implementation has a small impact on the overhead response time. When we look at the experiments with middle and the highest priorities, the difference between the response times of the two implementation is noticeable. The proposed implementation outperforms the reference implementation.

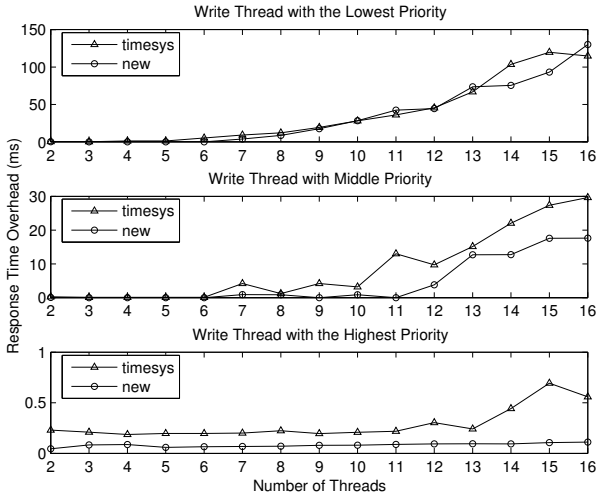


**Figure 11. Response Time Overheads of Write Tasks with Different priorities**

There are more experimental results with a similar trend in [5]. We also implemented the `WaitFreeReadQueue` class for RTSJ and performed extensive experiments to compare with the reference implementation. We obtained similar results. The results can also be found at [5].

## 6 Conclusion

Efficient implementations of the RTSJ queue classes are presented in this paper. The wait-free queue classes proposed in the Real-time Specification for Java are of general interest to any real-time synchronization system where hard real-time tasks have to synchronize with soft or even non real-time tasks. The implementations presented here are designed with the unidirectional nature of these queues in mind. In the implementation, we introduce a new solution to the "enabled late-write" problem. The new solution is

more efficient, with respect to space, compared to previous solutions without losing in time complexity.

The proposed implementation have several advantages. First, the implementation supports multiple real-time threads to access the queue in a wait-free manner at the same time. The reference implementation from Timesys only support one real-time threads to access the queue in a wait-free manner at the same time. Second, only read and write memory operations are used in our implementation. Our implementation keeps the "Write Once, Run Anywhere" principle and can be run in all environments which support RTSJ. Third, our implementation is efficient. Experiments shows that our implementation outperforms the reference implementation, especially when it comes to high priority tasks.

There are several ways that future research in wait-free synchronization can contribute to real-time Java. Very promising, we believe, is the investigation of practical wait-free implementations of garbage collection in the RTSJ model. The garbage collector is a central component of the Java environment. A wait-free implementation will improve the programmers ability to correctly reason about the temporal behavior of their Java programs.

## References


[1] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 111–122. IEEE, Dec. 1997.

[2] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC '97)*, pages 229–238. ACM, Aug. 1997.

[3] G. Bollella and J. Gosling. The real-time specification for java. *IEEE Computer*, 33(6):47–54, June 2000.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000. URL: `www.javaseries.com/rtj.pdf`.

[5] D. Cederman and T. Dellsén. A study of queue algorithms for wait-free inter-thread communication in real-time java. Master's thesis, Department of Computer Science, CHALMERS UNIVERSITY OF TECHNOLOGY, 2005. http://www.dtek.chalmers.se/~d00ceder/thesis/index.html.

[6] P. Dibble and A. Wellings. The real-time specification for java: current status and future work. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 71–77. IEEE Computer Society Press, 2004.

[7] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[8] M. Higuera-Toledano. Illegal references in a real-time java concurrent environment. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 321–324. IEEE Computer Society Press, 2004.

[9] S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal system support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 233–242. ACM, May 1996.

[10] TimeSys. The java reference implementation (RI) for real-time specification for java. http://www.timesys.com/products/java/, 2005.

[11] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01)*, pages 134–143. ACM, July 2001.

[12] A. Wellings and A. Burns. Asynchronous event handling and real-time threads in the real-time specification for Java. In *Proceedings of the Eighth Real-Time and Embedded Technology and Applications Symposium*, pages 81–89. IEEE Computer Society Press, 2002.

[13] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251. IEEE Computer Society Press, 2004.