

## Efficient Adaptations of the Non-Blocking Buffer for Event Message Communication

K. H. (Kane) Kim and Juan A. Colmenares\*

Department of Electrical Engineering and Computer Science  
University of California, Irvine, USA  
{khkim, jcolmena}@uci.edu

Kee-Wook Rim

Division of Computer and Information Sciences  
Sunmoon University, Korea  
rim@sunmoon.ac.kr

### Abstract

*Enabling message communication among concurrent computing threads without relying on mutual exclusion (i.e., locking) is highly desirable in real-time computing systems. This paper presents a refined version of the Non-Blocking Buffer (NBB), which is a lock-free interaction mechanism that enables efficient event-message communication between a single producer thread and a single consumer thread. The NBB scheme presented here contains improvements over the previous version in two aspects. First, application designers now have the flexibility of choosing the consumer's retry strategy for the case when the buffer is empty but the producer is in the middle of inserting an item. Second, in the refined version the producer inserts pointers to data items into the buffer whereas the consumer obtains copies of the items. This design is consistent with the fact that shared heap management must be avoided to enable fully lock-free interaction between the producer and the consumer. This paper also discusses the approaches based on the NBB mechanism for supporting all conceivable producer-consumer scenarios.*

### 1 Introduction

In the community of designers and programmers of real-time computing systems there has been growing recognition of the importance of facilitating efficient message exchanges among concurrently executing components. The overall performance and timing predictability of real-time

computing systems are impacted in major ways by the message exchange mechanisms used. When the timing constraints are in scales of milliseconds or sub-milliseconds, rather than hundreds or thousands of milliseconds, safe and efficient low-level message communication mechanisms are of critical importance. Such cases include not only many real-time concurrent computing applications but also substantial parts of operating systems and middleware. In this paper, concurrently executing parts subject to such high-precision timing requirements are assumed to be structured as threads.

Preferably message-send and message-receive operations must be performed in manners avoiding the sinking of producer and consumer threads into blocked states. There have been a number of message communication mechanisms proposed over the past 20 years that do not rely on locking for well coordinated concurrent operations of producers and consumers. Some of them have been devised for a special type of message communication called *state message communication* (e.g., [6, 1]). In this case, whenever consumers become ready to check the message, they are only interested in processing the most up-to-date data from producers. Thus, there is a fixed memory location used to keep the latest available version of each state message, and consumers access that memory location in read-only fashion. Each new version of a state message from a producer overwrites the previous one regardless of whether consumers had read the previous version or not (i.e., state messages can be lost). In contrast, in *event message communication*, every message must be consumed (i.e., event messages cannot be lost). In this paper, we focus on event message communication.

One important realization that has been occurring in re-

---

\* Also with the Applied Computing Institute, Engineering Faculty, University of Zulia.

cent years is the possibility of using a circular event message buffer between a producer thread and a consumer thread without making the producer and the consumer experience blocked states. The Non-Blocking Buffer (NBB) [4], created by the first co-author, is a highly practical and efficient scheme that reflects such realization and enables harmonious buffer sharing. Since its creation the NBB framework has been further refined into a concrete optimized mechanism presented in this paper.

Compared to the only other non-blocking event message exchange scheme that uses a circular buffer and appeared in literature to our knowledge [8], NBB has the following advantages. First, a consumer thread can be designed to perform a retry strategy that better suits the specific characteristics of the real-time application under development. Secondly, its implementation does not require use of complicated atomically executing machine instructions other than simple integer write and integer read operations. Other proposed non-blocking schemes for exchanging event messages are based on the use of linked lists (e.g., [7]) and they incur higher overhead.

We also discuss several extended versions of the basic NBB mechanism that are highly efficient in accommodating multiple producers and consumers and yet yield tight access time bounds. Therefore, these approaches allow producers and consumers to run on the same processor or different processors of a shared-memory multiprocessor system without compromising the system's time predictability, which is an important attractive feature in highly reliable design of real-time computing systems.

Use of NBB in real-time concurrent programming is expected to grow rapidly in coming years. We have been replacing most of the critical sections in our middleware supporting real-time distributed computing objects [5, 2], Time-triggered Message-triggered Objects (TMOs) [3], with NBBs.

The rest of the paper is organized as follows. Section 2 discusses NBB in detail. Section 3 presents adaptations of the NBB mechanism for supporting all conceivable producer-consumer scenarios. Finally, the paper concludes in Section 4.

## 2 The Non-Blocking Buffer

The Non-Blocking Buffer (NBB) is a *circular FIFO queue* that facilitates the communication of event messages from a *single producer thread* (PROD) to a *single consumer thread* (CONS) without causing any party to experience blocking. NBB, abstractly depicted in Figure 1, is a variation of the traditional *circular buffer*. NBB provides two basic functions: i) `InsertItem`, which is called by PROD to deposit a data item into the buffer, and ii) `ReadItem`, called by CONS to obtain an item from the buffer. These

functions are shown in Figure 2 and described in detail in Section 2.1.

(1) NBB includes two counters, the *update counter* (UC) and the *acknowledgment counter* (AC), also called *ack-counter*. Both counters are used to ensure that PROD and CONS always access different slots in the circular buffer. The value of the update counter is interpreted as a pointer to the slot in the circular buffer to be used for accommodating the new data item in the next call of `InsertItem`. Similarly, the value of the ack-counter is interpreted as a pointer to the slot in the circular buffer that contains the item to be read in the next invocation of `ReadItem`. PROD and CONS read both counters, but only PROD modifies the update counter and only CONS modifies the ack-counter. Specifically, PROD increases the update counter during the insertion of an item into the buffer, and CONS increases the ack-counter when an item is read from the buffer. Also, the update counter and ack-counter are used to prevent PROD from inserting an item when the buffer is full and CONS from reading an item when the buffer is empty.

(2) It is essential that the update counter and ack-counter be of an *aligned single-word integer type*, thus reading or writing the counters is a trivially short *atomic* operation. Counter atomicity guarantees that every time PROD and CONS read a counter they obtain a valid value without requiring that any OS synchronization mechanism provide exclusive access to the counter. Therefore, PROD is a non-blocking writer for the update counter, CONS is a non-blocking writer for the ack-counter, and both PROD and CONS are non-blocking readers for the update counter and the ack-counter.

The actions taken by PROD to judge whether the buffer is full or not involve one non-blocking read of the ack-counter. Similarly, CONS reads the update counter to check if the buffer is empty or not and it does so without disturbing PROD in any way. In consequence, the NBB mechanism provides a blocking-free event-message buffer between PROD and CONS with practically negligible overhead for checking the status of the buffer.

(3) In order to avoid the buffer overflow, application designers should ensure that CONS invokes the function `ReadItem` at a frequency which is, on average, higher than or equal to the frequency at which PROD invokes `InsertItem`. In addition, the size of the NBB should be chosen so that PROD is allowed to generate temporary bursts of messages without saturating the buffer. Thus, CONS often finds the buffer empty whereas PROD finds the buffer full rarely. NBB was devised along with the adoption of the software design style of making PROD and CONS exit from the NBB when the buffer is found full and empty, respectively, and come back later to retry. However, it is up to the application designer when PROD and CONS retry.

(3.1) The update counter (UC) and the ack-counter (AC)

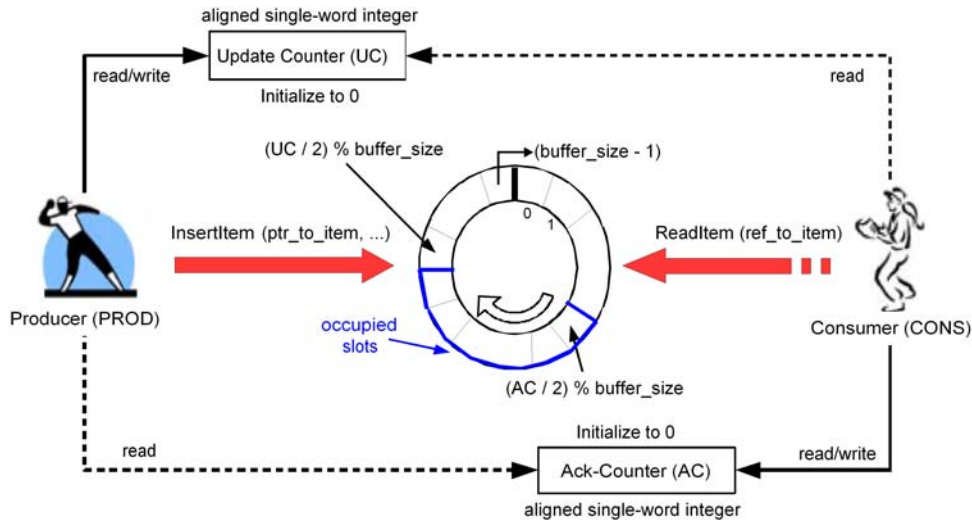


Figure 1. High-level view of the non-blocking buffer (NBB).

are increased twice during a successful execution of `InsertItem` and `ReadItem`, respectively. The update counter is increased by 1 just before `PROD` accesses the buffer for inserting an item. At this point the update counter is odd, which indicates that `PROD` is inserting an item into the buffer and has not finished yet. Immediately after inserting the item into the buffer, the update counter is increased by 1 again (i.e., now `UC` is even) to indicate that `PROD` finished to insert the item. By increasing the update counter in this way, the NBB's function `ReadItem` can detect and tell `CONS` that the buffer is empty but `PROD` is concurrently inserting an item. In this case application designers can use this knowledge to decide the *retry strategy* of `CONS` that better suits the specific characteristics of the real-time application under development. For example, `CONS` may keep retrying to read an item for a limited amount of time hoping that `PROD` will finish to insert the item by that time. Or `CONS` may sleep for some time and retry when it wakes up. Yet another option is for `CONS` to yield the processor to another thread and try again the next time it is scheduled. Considerations for the retry strategy of `CONS` are discussed in Section 2.2.

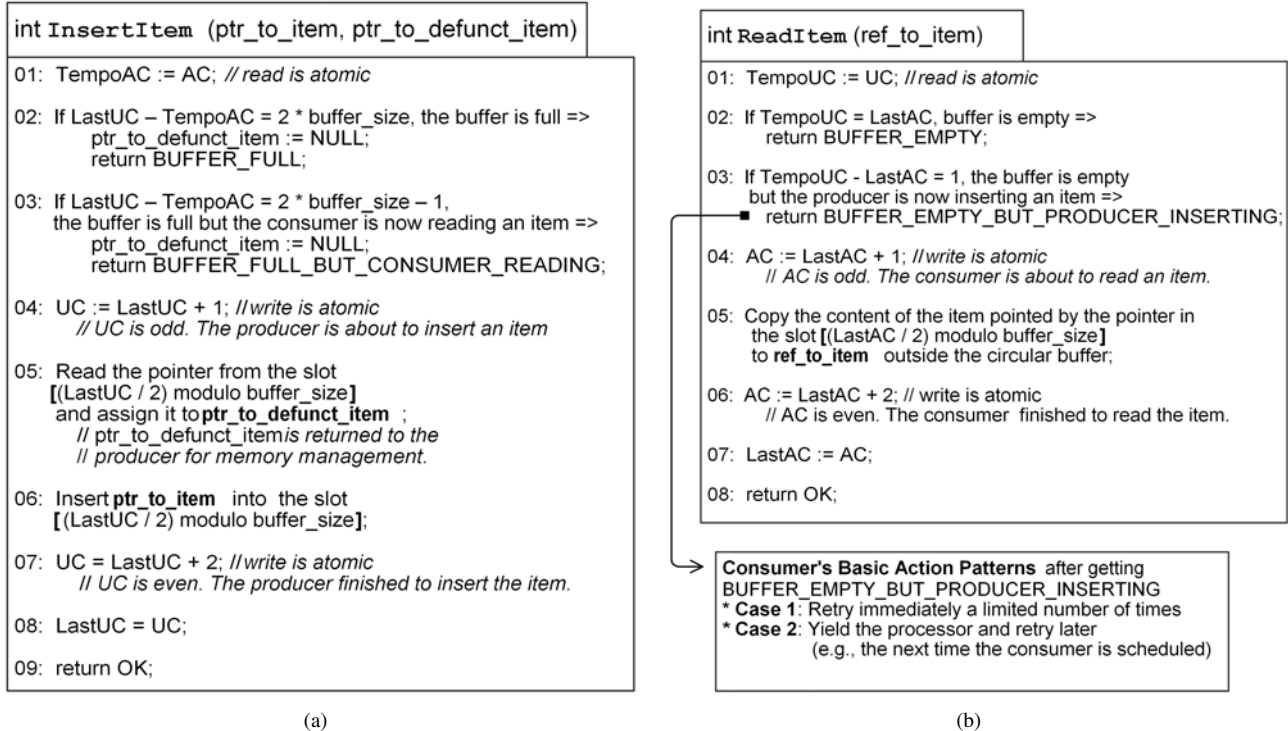
(3.2) Similarly, the ack-counter is increased by 1 just before and immediately after `CONS` reads an item from the buffer. An odd value in the ack-counter indicates that `CONS` is reading an item from the buffer, and an even value indicates that `CONS` already read the item. Thus, the NBB's function `InsertItem` can tell `PROD` that the buffer is full but `CONS` is concurrently reading an item. However, in this case this knowledge may not be for `PROD` as useful as it is for `CONS` in the case of the function `ReadItem` because application designers must guarantee that buffer saturation occurs very rarely during normal operation. Therefore, de-

signers may prefer to ignore the fact that `CONS` is in activity and define the actions that `PROD` will take only considering that the buffer is full.

(4) An important observation is that heap management shared by concurrent processes or threads cannot be freed of locking operations. Thus, even though NBB can communicate pointers to data items from `PROD` to `CONS` in a lock-free manner, dynamic memory management associated with the data items may result in locking operations involving `PROD` and `CONS`. Enabling fully lock-free interaction between `PROD` and `CONS` requires that they manage their own heap. In order to follow that design principle while minimizing the impact on efficiency, the NBB's functions `InsertItem` and `ReadItem` were defined such that `PROD` inserts pointers to data items into the buffer whereas `CONS` obtains copies of the items in memory owned by it. Thus, when `PROD` inserts a pointer to a data item into the NBB, `PROD` does not lose the ownership of that pointer. Instead, `PROD` just lends the NBB the pointer until `CONS` copies the item pointed by it. Then, the NBB must return the pointer to `PROD` so that the memory location associated with the item can be recycled or reclaimed. Therefore, NBB must be accompanied with mechanisms that allow `PROD` to obtain the pointers to items that have been already copied by `CONS`. In this paper those items are called *defunct items* and such mechanisms are discussed in Section 2.3.

## 2.1 Basic Functions of NBB

Figure 2 shows the details of the operation algorithms associated to the NBB mechanism. In this section we assume that the counters `UC`, `AC`, `LastAC`, and `LastUC` are



**Figure 2. Basic functions of NBB.**

initialize to zero, NBB stores pointers to data items in an array of size `buffer_size`, and the elements of the array are initialized to `NULL`.

The function `InsertItem` (Figure 2(a)) allows `PROD` to deposit a pointer to a data item into the NBB. This function determines which of the following cases occurs.

**Case I.1:** *The difference of `LastUC` and the ack-counter is equal to twice the size of the buffer* (line 02), where `LastUC` is the latest value of the update counter that `PROD` has produced. This means that the buffer is full and there is no evidence that `CONS` is reading an item from the buffer. Therefore, `InsertItem` returns `BUFFER_FULL` and `PROD` may try to execute the function later again as the application designer specified.

**Case I.2:** *The value of the ack-counter is an odd number and the difference of `LastUC` and the ack-counter is only one less than twice the size of the buffer* (line 03). It means that the buffer is full and `CONS` is in the middle of reading from the same buffer slot pointed by the update counter. Therefore, `InsertItem` returns `BUFFER_FULL_BUT_CONSUMER_READING` and `PROD` may try to execute the function later again as the application designer specified.

**Case I.3:** *The difference of `LastUC` and the ack-counter is less than twice the size of the buffer by two or more.* In this case there is room in the buffer for the new item

and thus, the function `InsertItem` proceeds to increase the update counter to indicate that `PROD` is about to insert the item. Next, the pointer in the buffer slot indicated by `LastUC` is assigned to `ptr_to_defunct_item` to be returned to `PROD`; thus, `PROD` can recycle or free the memory associated with the item pointed by that pointer. Then, the pointer to the new item is copied into the slot indicated by `LastUC`. The update counter is increased again to indicate that `PROD` already inserted the new item, `LastUC` is updated with the current value of the update counter, and finally, `InsertItem` returns `OK`.

On the other hand, `CONS` obtains a data item from the buffer by calling the function `ReadItem` (Figure 2(b)). Instead of returning a pointer to an item, `ReadItem` copy the requested item into memory allocated by `CONS`. The structure of `ReadItem` is very similar to that of `InsertItem`; thus, the following cases are considered.

**Case R.1:** *The update counter and `LastAC` are equal* (line 02), where `LastAC` is the the latest value of the ack-counter that `CONS` has produced. This means that the buffer is empty and there is no sign of `PROD` being in the middle of inserting an item. Therefore, `ReadItem` returns `BUFFER_EMPTY` and `CONS` may try to execute the function later again as the application designer specified.

**Case R.2:** *The value of the update counter is an odd num-*

ber and is larger than *LastAC* only by 1 (line 03). It means that the buffer is empty, but PROD is in the middle of writing a pointer to an item into the buffer slot next to the slot pointed by the ack-counter. Thus, `ReadItem` returns `BUFFER_EMPTY_BUT_PRODUCER_INSERTING` and CONS may try to execute the function later again as the application designer specified.

**Case R.3:** *The value of the update counter is larger than LastAC by two or more.* In this case the buffer is not empty, so the function `ReadItem` proceeds to increase the ack-counter to indicate that CONS is about to read an item. Next, the content of the item pointed by the pointer in the buffer slot indicated by *LastAC* is copied to the item referenced by `ref_to_item` and allocated by CONS. Then, the ack-counter is increased again to indicate that CONS already read the item, *LastAC* is updated with the current value of the ack-counter, and finally `ReadItem` returns OK.

## 2.2 The Consumer's Retry Strategy

There are different courses of actions that CONS may take when the return value of the NBB's function `ReadItem` indicates that the buffer is empty but PROD is in the middle of writing an item (line 03 in Figure 2(b)). Application designers have the flexibility of choosing for CONS the retry strategy that best suits the characteristics of the application in hand and the real-time computing platform used.

A key factor to be considered in devising a retry logic of CONS is whether PROD is likely to finish the insertion operation soon or not.

**Case RS.1:** *PROD and CONS run on different processors of a shared-memory multiprocessor system.* It is very likely that PROD will finish the insertion in a short time because it is a short operation (lines 05-07 in Figure 2(a)). Therefore, in this scenario it is reasonable that, when the return value of `ReadItem` is `BUFFER_EMPTY_BUT_PRODUCER_INSERTING`, CONS immediately starts checking the update counter until:

- i) the update counter becomes even, or
- ii) a predetermined short period of time, which should be a tight upper time bound for the PROD's insertion operation, i.e., `Max_Insertion_Time`, elapses.

In case i), CONS calls `ReadItem` again and will obtain the item that PROD just inserted. In case ii), i.e., if `Max_Insertion_Time` elapses, it is an indication that PROD lost the necessary execution resources to another thread without completing the execution of `InsertItem`. In this case, it is not effective for CONS to continue checking the update counter; hence, CONS should exit the retry loop and try to execute `ReadItem` again later (as the application designer specified).

**Case RS.2:** *PROD and CONS run on the same proces-*

*sor.* Here the *scheduling discipline* influences the decision about the retry strategy of CONS. In particular, if PROD and CONS are scheduled by a simple time-slicing round-robin, a fixed-priority, or an earliest-deadline-first (EDF) scheduler, and if the function `ReadItem` invoked by CONS returns `BUFFER_EMPTY_BUT_PRODUCER_INSERTING`, then it is not possible for PROD to complete the insertion while CONS is executing. For example, this situation arises if the PROD's time-slice expires when PROD has increased the update counter to an odd number, and the CONS's time-slice becomes active under a simple time-slicing round-robin scheduler. Therefore, it is not effective for CONS to retry immediately. For example, CONS may yield the processor to another thread and try again the next time it is scheduled.

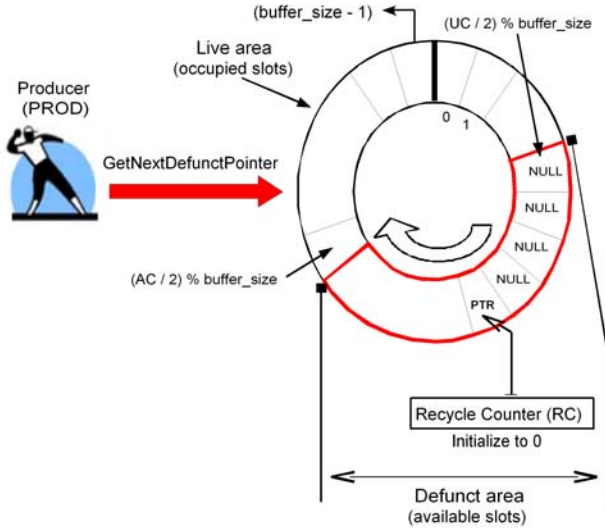
## 2.3 Returning Pointers to Defunct Items to the Producer

An aspect that differentiates the previous version of NBB [4] from the version presented here is that the latter explicitly indicates that PROD inserts pointers to data items into the buffer and CONS obtains copies of the items. This is in agreement with the fact that in order to avoid locking operations, not caused by the NBB mechanism itself but by shared dynamic memory management, PROD and CONS must not share the heap.

When PROD inserts a pointer into the NBB, PROD just lends the NBB the pointer, and once CONS copies the item pointed by the pointer, PROD must obtain the pointer back to reuse the memory area associated with the item. Therefore, NBB must provide ways for PROD to obtain pointers to items that have been already copied by CONS (i.e., pointers to *defunct items*).

One way is provided by the function `InsertItem`. `InsertItem` first reads the pointer from the buffer slot indicated by the update counter before writing the pointer to the new item into the same slot (line 05 in Figure 2(a)). Then, the pointer that was in the slot is returned to PROD. If the returned pointer is NULL, then either the buffer slot had not been used before, or the pointer that was in the buffer slot was already returned to PROD by other means. Otherwise, the returned pointer points to the memory area occupied by the defunct item that PROD can dispose of.

One problem with `InsertItem` is that PROD starts receiving pointers to defunct items only after inserting the first `buffer_size + 1` items. This is not desirable because `buffer_size` is rarely small and PROD has to keep `buffer_size + 1` items in memory all the time even if CONS has read most of them. To overcome this inconvenience NBB also provides the function `getNextDefunctPointer` that allows PROD to immediately obtain a pointer to a defunct item, if such pointer



**Figure 3. Basic operation of the function getNextDefunctPointer.**

exists in the buffer.

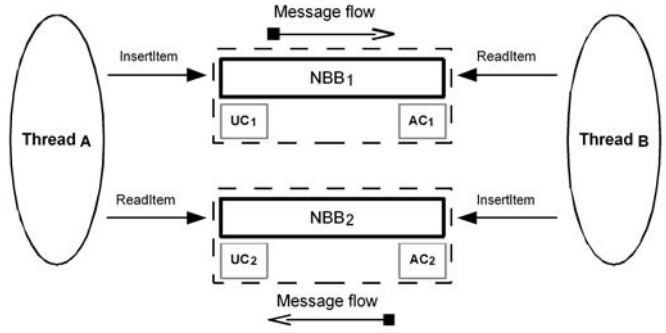
Figure 3 illustrates the basic operation of getNextDefunctPointer. We distinguish two areas in the circular buffer:

1. The *defunct area*, which comprises the buffer slots that are available; they are the slots clockwise ranging from the slot pointed by the update counter to one slot before the slot pointed by the ack-counter.
2. The *live area*, which comprises the buffer slots that are not available; they are the slots clockwise ranging from the slot pointed by the ack-counter to the one slot before the slot pointed by the update counter.

The function getNextDefunctPointer introduces a third counter, called the *recycling counter* (RC), that keeps track of the pointers to defunct items that have been returned to PROD. After verifying that the recycling counter points to a valid buffer slot in the defunct area, getNextDefunctPointer reads the pointer in the slot indicated by the recycling counter, writes NULL into the slot to indicate that the pointer has been returned to PROD, increases the recycle counter by 1, and finally returns the pointer to PROD. Thus, PROD can obtain all pointers to defunct items by repeatedly using getNextDefunctPointer.

### 3 NBB-based Approaches for Different Producer-Consumer Scenarios

NBB mechanism only provides lock-free communication of event messages between a single producer thread



**Figure 4. Symmetrical connections between threads via two NBBs.**

and a single consumer thread. However, extended versions of NBB can be developed to support more complex scenarios. This section presents NBB-based approaches for all conceivable producer-consumer scenarios.

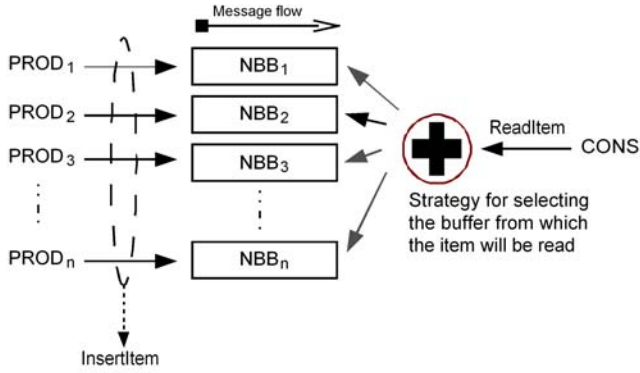
#### 3.1 Two-way Interaction via NBBs Between Two Threads

If the relationship between two threads is symmetrical, i.e., each thread functions as a producer as well as a consumer, two NBBs can be used between the two threads, each providing a one-way event-message transfer path. Figure 4 illustrates such two-way data paths between threads.

#### 3.2 Multiple-Producers/One-Consumer (\*P/1C)

This scenario consists of a number of producers ( $PROD_1, \dots, PROD_n$ ) that send event messages to a single consumer (CONS). The NBB-based approach adopted for this scenario is shown in Figure 5 and consists of multiple NBBs, one NBB per producer, and a *strategy* for determining the NBB from which CONS must obtain a data item when it invokes the function ReadItem.

Since each producer inserts items into a different NBB, from the producers' viewpoint this scenario is equivalent to the basic one-producer/one-consumer scenario (Section 2). On the other hand, the behavior of CONS depends on the strategy for selecting the NBB to be read. Different strategies may be considered for this purpose. The simplest one is round-robin. It makes CONS iterate over empty NBBs until a non-empty NBB is found and an item is extracted from it. If all the NBBs are empty the operation fails and returns BUFFER\_EMPTY to CONS. Thus, in the worst case the round-robin strategy is  $O(n)$  where  $n$  is the number of producers and NBBs.



**Figure 5. NBB-based approach for multiple producers and a single consumer.**

The round-robin strategy makes CONS consume messages in a different order from which they were inserted, but we believe that it is a minor disadvantage that does not affect most of the applications that fit this scenario. A more complex strategy must be employed if we are required to preserve the order of the messages. A possible solution is to add to every message a timestamp of when the message was inserted and use a priority queue to order the messages by timestamps before they are retrieved. Whenever CONS invokes ReadItem, it iterates over the NBBs, extracts an item from each non-empty NBB, and inserts the item into the priority queue. Once all the items obtained from the non-empty NBBs are inserted into the priority queue, the item from the top of the priority queue (i.e., the item with the earliest timestamp) is obtained and returned. The complexity of this solution is  $O(n \log n)$ .

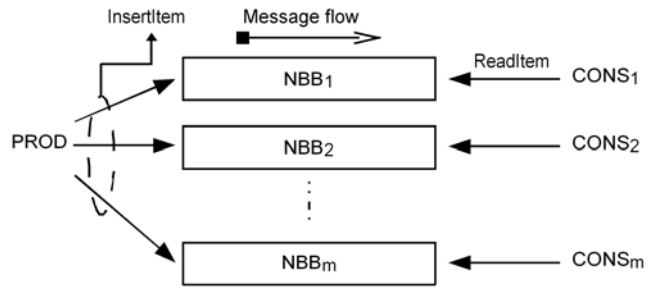
Because of its simplicity, we adopted the round-robin strategy for this scenario. Note that no new blocking possibilities are introduced.

### 3.3 One-Producer/Multiple-Consumers (1P/\*C)

This scenario consists of *one single producer* (PROD) that sends event messages to  $m$  consumers ( $CONS_1, \dots, CONS_m$ ). It can be divided further into *four sub-scenarios* that relate to different modes of communicating event messages between PROD and  $CONS_1, \dots, CONS_m$ . The sub-scenarios are described in the following sections.

#### 3.3.1 Unicast Communication (1P/\*C-Unicast)

Here PROD sends different and independent messages to each consumer  $CONS_j$  (i.e., for PROD each  $CONS_j$  is different). In this case, use of multiple NBBs is inevitable. The



**Figure 6. NBB-based approach for unicast communication between one producer and multiple consumers.**

adopted approach, shown in Figure 6, consists of one dedicated NBB between PROD and each consumer  $CONS_j$ .

#### 3.3.2 Broadcast Communication (1P/\*C-Broadcast)

In this case the producer (PROD) sends the same message to *all* the consumers. The adopted approach for this sub-scenario, shown in Figure 7, is a simple extension of the basic NBB mechanism discussed in Section 2. It consists of a single circular buffer shared by PROD and all the consumers  $CONS_1, \dots, CONS_m$ . Associated with each consumer  $CONS_j$  there is an ack-counter ( $AC_j$ ), and associated with PROD is the update counter (UC). To obtain an item from the buffer, each consumer  $CONS_j$  proceeds as usual accessing its own ack-counter (i.e.,  $AC_j$ ). On the other hand, when inserting a pointer to an item, PROD reads all the  $m$  ack-counters and chooses the ack-counter closest to the update counter for determining whether the buffer is full or not.

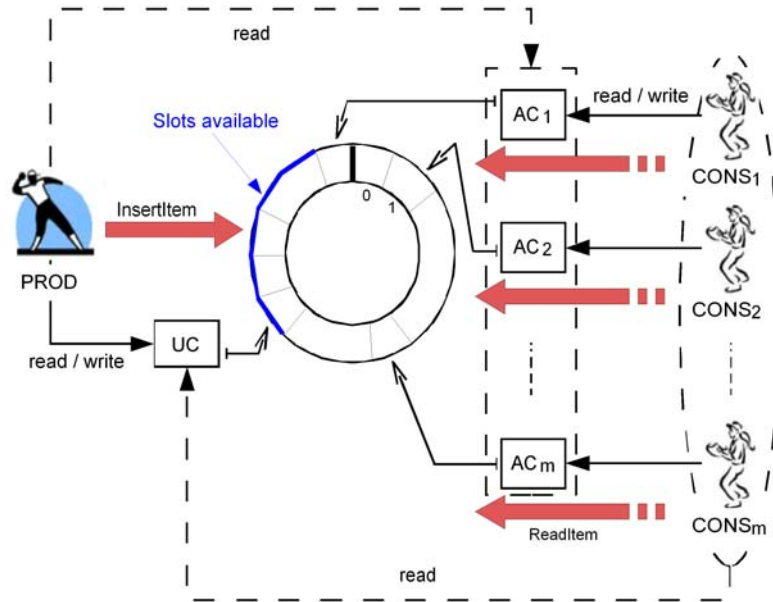
Because multiple consumers share a single buffer and copy the same data items, an item is considered defunct and can be returned to PROD only after all the consumers have read it.

Note that the approach for the 1P/\*C-Unicast scenario (Section 3.3.1) can also be used to support the current scenario, but it requires  $m$  NBBs (i.e., one per consumer) and PROD must perform  $m$  insertions per item. Instead, the solution proposed here only requires one buffer and PROD only inserts the item once, although PROD reads and compares  $m$  ack-counters.

#### 3.3.3 Multicast Communication (1P/\*C-Multicast)

In this sub-scenario PROD sends the same message to various consumers, but not to all of them. Moreover, different messages may be for different groups of consumers.

One possibility is to extend further the approach for the 1P/\*C-Broadcast sub-scenario (Section 3.3.2) by adding to



**Figure 7. NBB-based approach for broadcast communication between one producer and multiple consumers.**

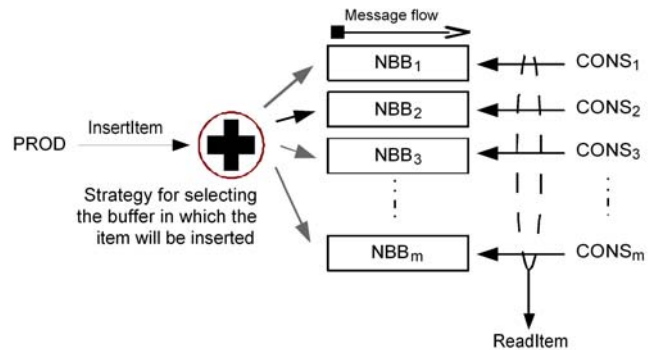
each message a list with the IDs of the consumers that must receive the message. When a consumer  $CONS_j$  tries to read an item from the buffer,  $CONS_j$  first needs to iterate over the list of receivers to check if it is in the list. If not, it must ignore the message, grab the next one from the buffer and do the same.  $CONS_j$  must repeat this procedure until finding that it is the receiver of a message or that there is no message for it in the buffer. Clearly, in the worst-case this operation is quadratic.

A better alternative is to apply the approach for the 1P/\*C-Unicast sub-scenario (Section 3.3.1) because it requires no modifications and consumers determine whether there is a message for them or not in constant time (i.e.,  $O(1)$ ), instead of in quadratic time. The only disadvantage is that PROD must insert the item in different NBBs.

### 3.3.4 Anycast Communication (1P/\*C-Anycast)

Here any consumer, but only one, can read a message sent by the producer (PROD). Once a consumer starts reading a message, the message is not available for the rest of consumers.<sup>1</sup> From the producer's viewpoint, the consumers ( $CONS_1, \dots, CONS_m$ ) are not different from each other.

Figure 8 shows the adopted approach for this sub-scenario. It consists of multiple NBBs, one NBB per consumer, and a strategy for determining in which NBB PROD



**Figure 8. NBB-based approach for anycast communication between one producer and multiple consumers.**

should insert the data item.

Since each consumer reads from a different NBB, from the consumers' viewpoint this scenario is equivalent to the basic one-producer/one-consumer scenario (Section 2). On the other hand, the behavior of PROD depends on the strategy for selecting the NBB for inserting the item. There are a number of possible strategies. In order to make our choice, we take advantage of the fact that in order to avoid buffer overflow consumers must read items at a higher frequency than that at which PROD inserts items. This guaran-

<sup>1</sup>Note that the term "anycast" refers to a different concept than that used in network communication.

tees that, in normal conditions, buffers are rarely full. Thus, a simple round-robin strategy will usually be able to insert an item into the very next NBB with practically negligible overhead.

### 3.4 Multiple-Producers/Multiple-Consumers (\*P/\*C)

This scenario consists of  $n$  producers ( $PROD_1, \dots, PROD_n$ ), which send data to  $m$  consumers ( $CONS_1, \dots, CONS_m$ ). Like the 1P/\*C scenario (Section 3.3), it can be divided into *four sub-scenarios* that relate to different modes of communicating event messages between the producers and the consumers. The sub-scenarios are described in the following sections.

#### 3.4.1 Unicast Communication (\*P/\*C-Unicast)

Here each producer  $PROD_i$  sends independent messages to each consumer  $CONS_j$  (i.e., for each  $PROD_i$  each  $CONS_j$  is different). In this case, we replicate the solution for the 1P/\*C-Unicast sub-scenario (Section 3.3.1) for each producer  $PROD_i$ . Thus, there is one dedicated NBB between each producer  $PROD_i$  and each consumer  $CONS_j$ . Each consumer reads from the NBBs in round-robin fashion.

#### 3.4.2 Broadcast Communication (\*P/\*C-Broadcast)

In this case each producer  $PROD_i$  sends the same message to *all* the consumers. The adopted approach for this sub-scenario is to replicate the solution for the 1P/\*C-Broadcast sub-scenario (Section 3.3.2) for each producer  $PROD_i$ . Therefore, each producer  $PROD_i$  shares its buffer with all the consumers. Each consumer reads from the NBBs in round-robin fashion.

#### 3.4.3 Multicast Communication (\*P/\*C-Multicast)

In this sub-scenario each producer sends the same message to various consumers, but not to all of them. Moreover, different messages may be for different groups of consumers. In this case we adopt the same approach for the sub-scenario \*P/\*C-Unicast (Section 3.4.1). Note that this is equivalent to replicate the solution for the sub-scenario 1P/\*C-Multicast (Section 3.3.3) for each producer.

#### 3.4.4 Anycast Communication (\*P/\*C-Anycast)

Here any consumer, but only one, can read a message sent by any producer. Once a consumer reads a message, the message is not available for the rest of consumers.

The adopted approach for this sub-scenario is shown in Figure 9 and can be seen as the combination of the approaches for the \*P/1C scenario (Section 3.2) and the

1P/\*C-Anycast scenario (Section 3.3.4). It consists of two sets of NBBs: one for producers (left) and the other for consumers (right). Each producer and each consumer are assigned an NBB from their corresponding set. A *broker* is in the middle of both NBB sets. The function of the broker is to retrieve pointers to data items from the producers' NBBs and inserts the pointers into the consumers' NBBs in round-robin fashion. The broker also is responsible of returning the pointers to defunct items to their corresponding producers. The broker may be a dedicated thread. Instead, we allow the consumers to play that role when needed; thus, our solution has a *virtual broker*. Whenever a consumer  $CONS_j$  finds its NBB empty, it tries to become the broker. First,  $CONS_j$  tries to assign its ID to the variable `BrokerID`. If  $CONS_j$  succeeds, then it becomes the broker; otherwise,  $CONS_j$  considers the buffer empty and may try later again as the application designer specified. Thus, only one consumer can be the broker at a time.

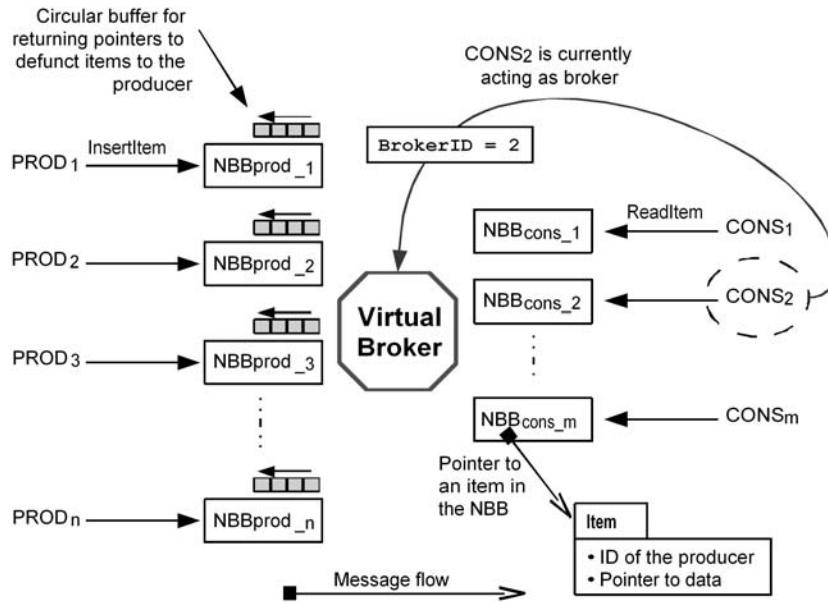
When  $CONS_j$  acts as the broker, it iterates over the producers' NBBs once. During this round  $CONS_j$  not only transfers pointers to items from the producers' NBB to its own NBB, but also to the NBBs of the other consumers. This operation is relatively quick because the broker only manipulates pointers, and by being *generous* with the other consumers,  $CONS_j$  contributes to reduce the average transfer time of the system.

At the same time,  $CONS_j$  obtains pointers to defunct items from the consumers' NBBs and insert each of those pointers into the circular buffer associated with the producer that inserted the pointer in first place (i.e., the producer that created the defunct item pointed by the pointer). In order for the broker to know which producer  $PROD_i$  is the owner of a particular pointer to a defunct item, the ID of the producer is additionally included in the data item. Note that this way of returning the pointers to defunct items to the producers differs from that of the basic NBB mechanism.

## 4 Conclusion

In this paper, we presented a refined version of the Non-Blocking Buffer (NBB), which is a simple interaction mechanism that enables efficient lock-free event-message communication between a single producer and a single consumer. The fundamental requirement for implementing NBB is that the *update counter* and the *acknowledgment counter* (i.e., the counters that control the access of the producer and the consumer to the slots of the circular buffer) must be of an *aligned single-word integer* type. Thus, reading and writing those counters are atomic operations.

Two aspects differentiate the NBB scheme presented here from the previous scheme discussed in [4]. First, the current scheme gives application designers the flexibility of choosing the consumer's retry strategy for handling the case



**Figure 9. NBB-based approach for anycast communication between multiple producers and multiple consumers.**

when the buffer is empty but the producer is in the middle of inserting an item. In contrast, the previous NBB scheme includes a fixed retry logic for that case. Second, unlike the previous one, the current NBB scheme recognizes that in order to enable fully lock-free interaction between the producer and the consumer, shared dynamic memory management must be avoided and the producer and the consumer must have separate heaps. Therefore, the NBB was designed so that the producer inserts pointers to data items into the buffer whereas the consumer obtains copies of the items. Additionally, the NBB also provides ways for the producer to obtain pointers to items that have been already copied by the consumer, thus the producer can properly manage the memory allocated for those items.

In this paper, we also described NBB-based approaches for supporting the following producer-consumer scenarios: i) two-way interaction between two threads, ii) multiple producers and one consumer, iii) one producer and multiple consumers, and iv) multiple producers and multiple consumers. Moreover, the last two scenarios were divided into 4 sub-scenarios that represent the different ways of communicating event messages between the producer(s) and the consumers (i.e., unicast, broadcast, multicast, and anycast). Those approaches do not introduce new blocking situations nor impose restrictions on the relative location of the producers and consumers.

NBB has turned out to be particularly valuable in building TMOSM (Time-triggered Message-triggered Object Support Middleware) [5, 3, 2], which is a middleware ar-

chitecture for supporting real-time distributed objects that we have formulated, experimented with, and discussed in literature in recent years. Although some experiments have been conducted with the NBB, much further experimental studies are needed to obtain better understanding of the potential of the mechanism. Also, the impacts of using NBB as interaction mechanism on resource allocation techniques within operating systems and middleware are considered an important subject for future research.

## References

- [1] J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Proc. of the 10th Euromicro Workshop on Real-Time Systems*, pages 2–9, June 1998.
- [2] S. F. Jenks, K. Kim, E. Henrich, Y. Li, L. Zheng, M. H. Kim, H.-Y. Youn, K. H. Lee, and D.-M. Seol. A Linux-based implementation of a middleware model supporting time-triggered message-triggered objects. In *Proc. of the 8th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 350–358, May 2005.
- [3] K. H. K. Kim. APIs for real-time distributed object programming. *IEEE Computer*, 33(6):72–80, 2000.
- [4] K. H. K. Kim. A non-blocking buffer mechanism for real-time event message communication. *Real-Time Systems*, 32:197–211, 2006.
- [5] K. H. K. Kim, M. Ishida, and J. Liu. An efficient middleware architecture supporting time-triggered message-triggered objects and an NT-based implementation. In *Proc. of the 2nd*

*IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, pages 54–63, May 1999.

- [6] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: a solution to a real-time synchronization problem. In *Proc. of the 14th IEEE Real-Time Systems Symposium*, pages 131–137, December 1993.
- [7] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.
- [8] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proc. of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*, pages 134–143, 2001.