Wait-Free Objects for Real-Time Systems?
(*Position paper***)**

Michel RAYNAL
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
raynal@irisa.fr

Abstract

The aim of this position paper is to promote the use of wait-free implementations for real-time shared objects. Such implementations allow the nonfaulty processes to progress despite the fact the other processes are slow, fast or have crashed. This is a noteworthy property for shared real-time objects. To assess its claim, the paper considers wait-free implementations of three objects: a renaming object, an efficient store/collect object, and a consensus object. On an other side, the paper can also be seen as an introductory survey to wait-free protocols.
Keywords*: Fault-Tolerance, Graceful Degradation, Real-Time, Wait-Free Objects.*

1 Introduction

"*A real-time system is a system which must produce outputs within deadlines in response to stimuli from application environments or spontaneously within specified time intervals*" [14]. Hence, real-time systems have strong timeliness and dependability requirements. As a consequence, the sharing of objects by real-time processes is a crucial issue. Usually, accesses to shared objects are protected by locking mechanisms (e.g., mutual exclusion). As, at any given time, a single process can access the object, the object state remains consistent despite concurrency. Unfortunately this commonly used approach presents an inherent serious drawback with respect to fault-tolerance: if a process crashes while it is accessing an object protected by a lock, the other processes are prevented from accessing the object [2]. The system has then to detect the process crash, re-establish a consistent object state and finally allow the requesting processes to access the object. This is difficult and time consuming. So, a natural question is the following "*How to prevent this bad situation to occur despite process crashes?*". This paper considers wait-free computing to answer this question.

Wait-free computing has been introduced in 1977 by Lamport [17]. Since then, (mostly on the theoretical side) several problems have been solved with simple and elegant wait-free protocols (e.g., [23]). An implementation of an object is wait-free if every access by a nonfaulty process is guaranteed a response regardless of whether the other processes are slow, fast or have crashed [12]. This means that a nonfaulty process that invokes an operation on a wait-free object always gets a correct answer. Its progress cannot be prevented by the other processes. Moreover, according to the object, the maximal duration of an operation on a wait-free object can be bounded. This can allow a system designer to meet real-time system requirements[1].

This paper considers a multiprocessor system and presents in an increasing order of difficulty wait-free implementation of three objects: a renaming object, a store/collect object and a consensus object. More sophisticated wait-free protocols can be found in the literature (e.g., the wait-free implementation of a snapshot object [5]). The aim of this paper (which has a spirit similar to [25]) is to introduce and promote the use of wait-free implementations in the design of real-time objects. It can also be seen as a simple introduction to wait-free objects.

The paper is made up of six sections. Section 2 introduces the computation model. Then the next three sections provide wait-free solutions to the mentioned problems. The interested reader can find deeper developments on wait-free computing in [8, 11, 12]. Finally, Section 6 discusses wait-free computing for real-time applications.

[1] (A few *lock-free* objects for uniprocessor real-time systems have been designed [16]. While wait-free objects guarantee that their operations cannot starve, lock-free objects provide a weaker guarantee: they only ensure that their operations do not deadlock. But, as shown in [2], the use of appropriate scheduling rules to lock-free object operations can prevent their starvation in an uniprocessor system.

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

2 Computation Model

2.1 Atomic Read/Write Shared Memory

For most of the paper we consider a standard asynchronous shared-memory system with n processes ($n > 1$), where at most f ($0 \leq f < n$) may crash. A nonfaulty (or correct) process is a process that never crashes. A faulty process executes correctly (i.e., according to its specification) until it crashes. After having crashed, a process executes no operation. Hence, after crashing, a process does not modify variable states.

The shared memory consists of multi-writer/multi-reader atomic registers (also named shared variables). A process p_i can have local variables: those are private in the sense p_i is the only process that can read or write them. When needed, the subindex i is used to denote p_i's local variables. For more details on the computation model see any standard textbook.

2.2 Consistency Criterion: Linearizability

Processes cooperate by accessing shared objects. Each object has a type which defines the set of its possible values and a set of operations that provide the only means to manipulate it.

In a sequential system, the object operations are invoked one after the other. Consequently, the meaning of each operation can be given by a pair of predicates (pre- and post-condition). In the following, we are interested in objects that have a state. This means that the result of an operation may depend on the sequence of the past operations that accessed the object.

In a concurrent system, object operations can be concurrently invoked by processes. Hence, it is necessary to define which operation interleavings are correct. That is the aim of the linearizability notion. Intuitively, a concurrent (i.e., shared) object is linearizable if it behaves as a correct sequential object. Linearizability is a correctness criterion for concurrent objects that has been formally defined in [10]. It extends to arbitrary objects the atomicity notion currently used for atomic read/write registers. More precisely, a shared object is *linearizable* if any execution of its operations appears as if these operations were executed (1) sequentially and in agreement with the sequential specification of the object, and (2) in an order compatible with their real-time occurrence order.

Linearizability implicitly considers *unary* objects, i.e., for any object type T, an operation on a T type object involves a single object of that type. There exist correctness criteria for *n-ary* concurrent objects (e.g., *Normality* [7]). In the following we only consider unary objects.

3 A Renaming Object

3.1 The M-Renaming Problem

Let us assume that the n processes $p_1, \ldots, p_n$ have arbitrarily large (and distinct) names $id_1, \ldots, id_n$. In the (one-time) M-renaming problem, the processes are required to get new names in a way such that the new names belong to the set $\{0, \ldots, M-1\}$, and no two processes get identical names [3].

The problem consists in designing a persistent object providing an operation (get_name) that allows processes to get such new names. We present a simple solution (due to Moir and Anderson [22]) where $M = n(n+1)/2$ (it has been shown that there is no solution if $M \leq n+f$ [9]). This protocol solves the one-time renaming problem, i.e., a process gets a name once for all. The *long-lived renaming* problem is characterized by two operations: get_name and release_name: a process uses a name only during a finite period and then releases it; a name that has been released can be used again by another process. Wait-free solutions to long-lived renaming can be found in [21, 22]. As we can see, although the renaming problem is a typical resource allocation problem, mutual exclusion-based solutions cannot tolerate process failures.

3.2 A Wait-Free Implementation

Moir-Anderson's wait-free solution is surprisingly simple and elegant. It uses a building block that (according to [4]) we call a *splitter*. This building block has been initially introduced by Lamport to provide fast mutual exclusion [18].

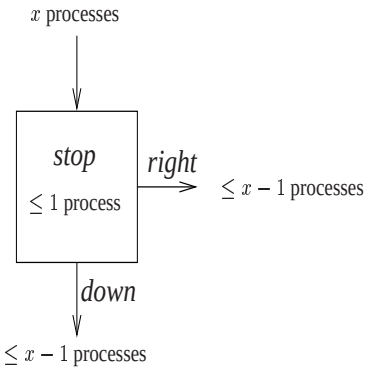

Figure 1. A Splitter

A basic building block A splitter is a concurrent object that assigns one out of three values ($stop$, $down$ or $right$) to a local variable $move_i$ of the invoking process

p_i. A splitter is characterized by the following global property: if x processes access the splitter, then at most one receives the value $stop$, at most $x-1$ receive the value $down$, and at most $x-1$ receive the value $down$ (Figure 1).

A wait-free implementation of a splitter is described in Figure 2. Its internal state is represented by two shared variables: X (will contain process ids) and Y (a boolean initialized to $false$). The procedure direction is the operation provided by the splitter to the processes; it returns its result in the local variable $move_i$ of the invoking process p_i.

```
procedure direction(move_i)

(1)  X ← id_i;
(2)  if Y then move_i ← right
(3)       else Y ← true
(4)            if (X = id_i) then move_i ← stop
(5)                          else move_i ← down
(6)  endif      endif
```

Figure 2. A Wait-Free Splitter Protocol

As there is no loop, the splitter is trivially wait-free. Let us assume that x processes access the splitter object. Let us first observe that, due to the initialization of Y, not all of them can get the value $right$ (for a process to obtain $right$, another process has to first set Y to $true$). Let us now consider the last process that executes line 1. If it does not crash, this process cannot get the value $down$ (due to line 4), hence, not all processes can get the value $down$. Finally, no two process can get the value $stop$. Let p_i be the first process that finds $X = id_i$ at line 4 (consequently, p_i gets the value $stop$ if it does not crash). This means that no process p_j has modified X while p_i was executing the lines 1-4. It follows that any $p_j \neq p_i$ that will modify X (at line 1) will find $Y = true$ (at line 2), and consequently cannot get the value $stop$.

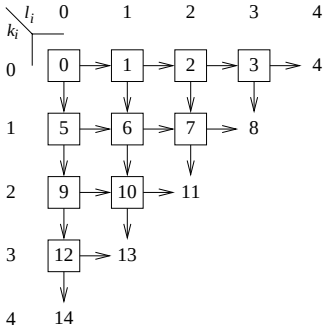

Figure 3. A Renaming Grid

Idea: a grid of renaming splitters The elegance and simplicity of Moir-Anderson's solution consists in a grid made up of $n(n-1)/2$ renaming splitters (Figure 3 depicts such a grid for $n = 5$). A process p_i first enters the left corner of the grid. Then, it moves along the grid according the values it obtains from the splitters ($down$ or $right$) until it gets the value $stop$. Finally, it considers as its new name the value associated with the splitter where it stopped. The property attached to each splitter ensures that no two processes stop at the same splitter.

The resulting Moir-Anderson protocol is described in Figure 4. $X[0..(n-2), 0..(n-2)]$ and $Y[0..(n-2), 0..(n-2)]$ are upper left triangular matrices of shared variables (the entries of Y are initialized to $false$). k_i, ℓ_i and $term_i$ are local variables of the invoking process p_i. The update $k_i \leftarrow k_i + 1$ (resp. $\ell_i \leftarrow \ell_i + 1$) implements a down move (resp. right move). The setting of $term_i$ to $true$ implements a stop. No process takes more than $(n-1)$ iteration steps. Moreover, as no two processes arrive at the same grid position after taking $(n-1)$ steps, the diagonal $(n-1, 0), (n-2, 1), \ldots, (0, n-1)$ of X and Y is not used. It is relatively easy to see that the worst case time complexity is $4(n-1)$ (maximum number of shared memory accesses).

```
function get_name(id_i)

(1)  k_i ← 0; ℓ_i ← 0; term_i ← false;
(2)  while ((k_i + ℓ_i < n − 1) ∧ ¬term_i) do
(3)      X[k_i, ℓ_i] ← id_i;
(4)      if Y[k_i, ℓ_i] then ℓ_i ← ℓ_i + 1 % move right %
(5)                     else Y[k_i, ℓ_i] ← true;
(6)                          if (X[k_i, ℓ_i] = id_i)
(7)                            then term_i ← true % stop %
(8)                            else k_i ← k_i + 1 % move down %
(9)                          endif
(10)     endif
(11) enddo;
(12) return(n × k_i + ℓ_i − (k_i(k_i − 1)/2))
         % the name is the position [k_i, ℓ_i] in the grid %
```

Figure 4. W-F $n(n+1)/2$ Renaming [22]

4 An Efficient Store/Collect Object

4.1 Definition: Adaptive Store/Collect Object

Let a *view* be a set of process-values pairs without process repetition, $V = \{\langle p_i, v_i\rangle, \ldots\}$. If $\langle p_j, v_j\rangle \in V$, then $val(p_j) = v_j$; $val(p_j) = \perp$ otherwise.

A store/collect object provides the processes with two operations. store(v) allows a process p_i to declare v as its latest value for a view; collect provides the invoking process with the view made up of the latest values

declared by the processes. Let us define a partial order on views as follows: $V1 \leq V2$ if for every process p_i such that $\langle p_i, v1\rangle \in V1$, then $\langle p_i, v2\rangle \in V2$ where the store($v2$) operation follows or is equal to the store($v1$) operation (note that both store operations are issued by p_i). Let collect$_1$ (resp. collect$_2$) an operation that gets $V1$ (resp. $V2$). If collect$_1$ terminates before collect$_2$ starts, then we require that $V1 \leq V2$ (collect$_1$ terminates before collect$_2$ if the last shared memory access issued by collect$_1$ precedes the first shared memory access issued by collect$_2$).

4.2 An Efficient Wait-Free Implementation

A trivial implementation of a store/collect object consists in using an array $V[1..n]$ whose ith entry keeps the last value declared by p_i. A simple implementation of collect consists in reading the array V. The time complexity of such a wait-free implementation is $O(n)$.

In some applications the actual number (k) of active processes (i.e., the processes that invoke store and collect) is not a priori determined and can be much smaller than the total number (n) of processes (i.e., $k << n$). Hence the idea to design a wait-free implementation of a store/collect object whose cost is $O(k)$ instead of $O(n)$. We only sketch here such an *adaptive* implementation due to Attiya *et al.* [4].

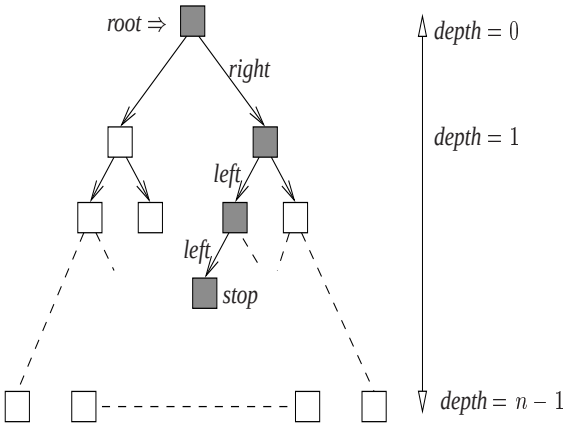

Figure 5. A Complete Bin Tree of Splitters

The implementation uses a complete binary tree T of depth $n-1$ (Figure 5). Only a subpart of the tree is actually used, but this part is not known in advance: it is dynamically defined according to the accesses issued by the processes. Interestingly, each node of the tree T contains a splitter (as defined in Figure 2).

The first store by a process p_i provides it with an available vertex vtx of the tree T where p_i can deposits its latest value. Then, the following store operations invoked by p_i will directly access the vertex vtx. To get an available vertex, a process traverses the tree from the root (accessed from the pointer $root$) towards a leaf. This traversal is determined by the values output by the splitters associated with the vertices that are visited, namely, the value $right$, $left$ or $stop$ ($left$ is used as synonym of $down$). A vertex is available when its splitter answers $stop$. Moreover, all the vertices accessed by a store operation are marked. This marking defines the subpart of the tree that is actually used. A collect by a process is a simple depth first traversal of the marked vertices of the tree: this visit collects the last values deposited by the processes.

A complete description of the protocol, its correctness proof and a complexity study can be found in [4]. Let k denote the current number of processes that have invoked a store or collect operation. Let us consider a process p_i. Its first store operation requires $O(k)$ shared memory accesses; its next store operations require $O(1)$ shared memory accesses. A collect operation requires $O(k)$ shared memory accesses. As we can see, the number of shared memory accesses for the first store operation of a process or a collect operation depends on the number of processes that have already started a store operation.

5 Consensus Objects

5.1 Why Consensus: Universality

A binary *consensus* object supports two operations propose_0 and propose_1. Each operation returns a value. The sequential specification of such a consensus object is the following: if propose_v is the first operation, then every operation returns the value v. (Said differently, consensus agreement states that no two processes get different values, and validity agreement states that a result value -also called decided value- is a value proposed by a process).

The consensus object is fundamental for a very simple reason. An object $\mathcal{O}$ is *universal* for a set S of concurrent objects if any object of S has a wait-free implementation from $\mathcal{O}$ and shared registers. Herlihy has shown the following fundamental result [8]: consensus is universal for any object that has a sequential specification. This is a strong result. To appreciate it, let us notice that objects made up of atomic shared registers that can be accessed with a test&set operation (or swap or fetch&add) are not universal. This means that these synchronization primitives are not powerful enough to construct wait-free implementation of objects whose specification is sequential (the interested reader in referred to Herlihy's seminal paper [8]).

The implementation of a consensus object is far from

being trivial. More precisely, it has been shown that such an object cannot be implemented in an asynchronous shared memory system where processes may crash [20]. This means that the underlying asynchronous system has to satisfy additional assumption for a consensus object be implementable.

5.2 Faulty Objects

The previous Sections 3 and 4 have considered the wait-free implementation of shared objects that tolerate the failure of processes. It has implicitly been assumed that the base objects on which these implementations rely are reliable (these base objects were mainly shared registers). In this section we consider that base objects may fail and we are interested in wait-free implementations that tolerate the failure of both processes and base objects. This problem has been addressed by Jayanti, Chandra and Toueg who have provided an in-depth study of the problem and designed a powerful family of wait-free protocols for fault-tolerant shared objects [12][2].

As consensus is an universal object, it is particularly interesting to get wait-free implementations of a consensus object that tolerate the failure of both processes and base objects.

Failure modes We consider that a (wait-free) consensus object can exhibit two types of failures: crash and omission (formal definitions can be found in [12]).

- A wait-free consensus object fails by *crash* if it behaves correctly until some time after which every propose_v operation returns the default value $\perp$. This means that the object behaves according to its specification until it crashes and then satisfies the property "once $\perp$, thereafter $\perp$". Hence, crashed operations are considered as having no effect.
- A wait-free consensus object fails by *omission* if (1) some propose_v operations return the default value $\perp$ (it is assumed that if a process gets the $\perp$ answer, it does not subsequently invoke operations on the object), and (2) when each operation returning $\perp$ is considered as a non-terminated (pending) operation, the object behavior agrees with its specification.

To take into account these failure modes, the consensus properties are modified as follows. Validity becomes: the value decided by a process is a proposed value or $\perp$. Agreement becomes: two processes that decide non-$\perp$ values decide the same value.

To understand the motivation and the subtleties of these definitions, let us consider a consensus object that fails by crash. This means there is a time t such that before t all operations get the same non-$\perp$ answer, and after t all operations get the same $\perp$ answer. If it fails by omission, some processes can get the answer $\perp$ (they experience an omission from the object) while the other processes get the same non-$\perp$ answer. Moreover, this non-$\perp$ answer can be the value v proposed by any process process p_j[3]).

It is easy to see that crash failures are also omission failures, but the converse is not true. That is why it is said that omission failures are strictly more severe than crash failures (and, more generally, arbitrary failures are strictly more severe than omission failures). This create a hierarchy of failure modes (see below).

Fault-tolerance and graceful degradation An implementation of a shared object is *t-fault tolerant with respect to a failure mode $\mathcal{F}$* (crash, omission, arbitrary) if the object remains correct and wait-free despite the occurrence of up to t base objects that fail according to $\mathcal{F}$. (Let us notice that if more than t base objects fail according to $\mathcal{F}$, nothing is required from the implementation.)

Let us say *the failure mode $\mathcal{F}$ is less severe than the failure mode $\mathcal{G}$* (denoted $\mathcal{F} \prec \mathcal{G}$) if a protocol that is t-fault tolerant for the failure mode $\mathcal{G}$ is also t-fault tolerant for the failure mode $\mathcal{F}$. We have: crashes $\prec$ omissions $\prec$ arbitrary failures.

Graceful degradation is a different but highly desirable property. A wait-free implementation of a shared object O is *gracefully degrading* if it never fails more severely than the base objects it is derived from, whatever the number of base objects that fail.

It is important to notice that the definition of the "severity" relation on failure modes ($\prec$) involves only the existence of a fault-tolerant protocol (it does not involve the notion of graceful degradation).

As an example let us assume that base objects can fail by crashing. If the implementation remains wait-free and correct despite the crash of any number of processes

[2] The reader inclined towards theory will find a formal presentation in [12]. The presentation that follows can be considered as a digest of [12, 13].

[3] This means that the decided value can be the value that has been proposed by a correct operation or a faulty operation. (It is important to notice that, when we consider the crash failure mode, a value decided by the processes has always been proposed by a propose_v operation that did not fail.) This is possible because the omission failure mode considers failed operations as non-terminated operations. Differently, the crash failure mode considers failed operations as having no effect.

and the crash of up to t base objects, then this implementation is t-fault tolerant with respect to the crash failure mode. If the implementation is wait-free and fails only by crash (if it fails) despite the crash of any number of processes and the crash of any number of base objects, then it is gracefully degrading. Of course, the ultimate goal is to design wait-free implementations that are t-fault tolerant and gracefully degrading.

5.3 Wait-Free t-Fault Tolerance

This section presents the wait-free t-fault tolerant implementation of a consensus object for omission failures introduced in [12]. This implementation is a self-implementation as the base objects are consensus objects. This is actually an appropriate "duplication" technique capable to cope with "duplicates" that fail. To prevent ambiguities, an operation on a base object is denoted propose_v, while an operation on the constructed object is denoted PROPOSE_v.

The implementation requires $t + 1$ copies of the base consensus object (which is optimal). These objects are kept in an array $base\_cons[1..(t + 1)]$. The protocol is described in Figure 6. Its underlying principle is simple. Each base object is a "copy" of the constructed object, and a PROPOSE_v operation sequentially visits all the base objects. As the protocol is required to cope with up to t base objects that suffer omission failures, it assumes that at least one base object is correct[4]. Let $base\_cons[k]$ be the first correct base object. It provides all the processes that invoke $base\_cons[k]$.propose_est (line 3) with the same non-$\perp$ value u (kept in the local variable est of the invoking process, line 4). Moreover, as each invocation of PROPOSE_v visits the base objects sequentially and in the same order (**for loop**, lines 2-5), the local variable est (which contains u) is not modified thereafter.

As shown in [12], it is worth noticing that this t-omission tolerant wait-free implementation is not gracefully degrading. Let us consider the case where (1) two processes p_i and p_j invoke PROPOSE_0 and PROPOSE_1 respectively, and (2) all base objects crash. As we can see, all $base\_cons[k]$.propose_est operations return $\perp$, and p_i gets 0 while p_j gets 1. The implementation is not correct as different values are returned to the processes (this corresponds to an arbitrary failure of the constructed object).

As this protocol is t-fault tolerant for the omission failures mode, according to the severity relation defined on failure modes ($\prec$), it is also t-fault tolerant for crash failures.

[4] According to the definition of "t-fault-tolerant protocol", let us remark that the protocol can behave arbitrarily when there are more than t base objects that fail.

```
procedure PROPOSE_v

% est, aux and k are local variables of the invoking process %
(1)  est ← v;
(2)  for k from 1 to (t + 1) do
(3)        aux ← base_cons[k].propose_est;
(4)        if (aux ≠ ⊥) then est ← aux endif
(5)  endfor;
(6)  return(est)
```

Figure 6. t-Fault Tolerant Consensus [13]

5.4 Wait-Free Graceful Degradation

The wait-free t-omission tolerant and gracefully degrading consensus implementation introduced in [13] is presented in Figure 7. It uses $2t + 1$ base objects (which has been shown to be optimal).

Similarly to the previous one, this protocol (1) is based on a duplication technique, and (2) visits the base objects sequentially. It has two main differences. The first is the fact it requires more base objects, namely $base\_cons[1..(2t + 1)]$. The second is its internal control structure, which is more intricate than the previous one. This is due to the fact that the protocol has to cope with up to t base objects failing by omission, AND, when more than t base objects fail by omission, it may exhibit omission failure but is not allowed to exhibit more severe failures (i.e., arbitrary failures).

The lines 4-6 are the crucial part of the protocol. If $base\_cons[k]$.propose_est returns a non-$\perp$ value d different from p_i's current estimate (est), p_i concludes that all base objects $base\_cons[1..(k - 1)]$ have failed. Accordingly, it changes est to $V[k] = d$, and sets $V[1..(k - 1)] \leftarrow [\perp, \ldots, \perp]$. It follows that, at the end of the loop, the local array V contains only $\perp$ and d (the current value of est). Finally, if no more than t entries of V are equal to $\perp$, p_i decides $est = d$, otherwise it decides $\perp$ (remark that p_i decides the majority value of its local array V). In the first case, the PROPOSE_v operation issued by p_i behaved correctly, while in the second case it suffered an omission failure. The second case can appear only if more than t base objects fail by omission.

5.5 Fault-Tolerance with Respect to Graceful Degradation

An important result proved in [12] states that it is impossible to design t-fault tolerant gracefully degrading wait-free implementations for the crash failure mode. It

```
procedure PROPOSE_v

% V[1..2t+1], est and k:
        local variables of the invoking process %
(1)  est ← v;
(2)  for k from 1 to (2t+1) do
(3)        V[k] ← base_cons[k].propose_est;
(4)        if (V[k] ≠⊥) ∧ (V[k] ≠ est)
(5)                then est ← V[k];
(6)                     V[1..(k−1)] ← [⊥, ..., ⊥]
(7)        endif
(8)  endfor;
(9)  if (the number of ⊥ in V is > t)
(10)       then return(⊥) else return(est) endif
```

Figure 7. Gracefully Degrading Cons. [12]

follows that there is no t-fault tolerant gracefully degrading self-implementation of consensus for crash failures. To give an intuition of this result, we show here that the particular protocol previously described in Figure 7 (which is t-fault tolerant gracefully degrading for omission failures) is not t-fault tolerant gracefully degrading for crash failures[5].

Let us consider the following scenario involving two processes p_i and p_j that invoke PROPOSE_0 and PROPOSE_1, respectively:
- p_i executes $base\_cons[1]$.propose_0 and gets 0,
- p_j executes $base\_cons[1]$.propose_1 and gets 0,
- then, p_j executes $base\_cons[k]$.propose_0 for $1 < k \leq 2t+1$ and gets 0 from each of them,
- Therefore, p_j exits, returning the value 0,
- Now, the $2t+1$ base objects fail (crash),
- p_i continues, proposing 0 to $base\_cons[k]$ ($2 \leq k \leq 2t+1$) and gets $\perp$ from each of them,
- finally, as $2t$ entries of p_i's vector are equal to $\perp$, p_i returns $\perp$.

In this execution, more than t base objects fail by crash, and the resulting object fail by omission. We show that this execution is not correct with respect to the crash failure mode. To be correct the execution must have a linearization that satisfies the consensus sequential specification in presence of crash. We show that this is not the case. If PROPOSE_0 by p_i is linearized before PROPOSE_1 by p_j, then as p_i returned $\perp$, p_j has to return $\perp$ ("once $\perp$, thereafter $\perp$" property of the crash failure mode), which is not the case. If PROPOSE_1 by p_j is linearized before PROPOSE_0 by p_i, then (according to the sequential specification of consensus) the value 1 has to be returned by p_j, which is not the case. Hence, in each case, there is something wrong.

[5] This is not counter-intuitive. Let us remind that the definition of the severity relation on failure modes ($\prec$) is based on fault-tolerance (but not on graceful degradation).

If an implementation is gracefully degrading for omission, base objects can experience "severe" failures (namely, omissions) but the implemented object is also allowed to experience a "severe" failure (omission). The fact that the implemented object can experience a severe failure (omission) makes the protocol design relatively simple. If an implementation is gracefully degrading for crash, base objects can experience "simple" failures (crashes), but the protocol has the obligation to ensure that the implemented object fails also in a "simple" way, namely by crash. And this is difficult to achieve. That is why combining fault-tolerance and graceful degradation is not possible for all failure modes.

6 Concluding Remarks

The aim of this position paper was to present wait-free implementations of shared objects. These objects have a persistent state that is kept consistent despite process crashes and base object failures.

We think that wait-free computing could be an interesting alternative to more traditional (lock-based) approaches currently used to implement real-time objects. This is motivated by the following simple reasons. First, wait-free computing copes naturally with process crashes. Second, an operation on a wait-free object has a bounded response time despite failures. Third, wait-free computing avoids priority inversion. Consequently, a wait-free object can harmoniously satisfy both timeliness and fault-tolerance constraints. This seems to be a first-class property for real-time objects [1, 15, 19, 24, 26].

To assess its claim, the paper has surveyed a few wait-free protocols implementing concurrent objects. In that sense, it can also be considered as an introductory step to wait-free computing.

Acknowledgment The author would like to thank Luiz Bacellar and Kane Kim for their help, and P. Jayanti for interesting exchanges on wait-free fault-tolerant objects.

References

[1] Anderson J.H., Jain R. and Ramamurthy S., Wait-Free Objects Sharing Scheme for Real-Time Uniprocessor and Multiprocessors. *Proc. 18th Int. IEEE Symposium on Real-Time Systems (RTS'97)*, pp. 111-121, 1997.

[2] Anderson J.H., Ramamurthy S. and Jeffay K., Real-Time Computing with Lock-Free Shared Objects. *ACM TOCS*, 15(2):134-156, 1997.

[3] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.

[4] Attiya H., Fouren A. and Gafni E., An adaptive Collect Algorithm with Applications. *Tech Report*, The Technion, Haifa, 2001.

[5] Attiya H. and Rachman O., Atomic Snapshots in O($n \log n$) Operations. *SIAM Journal of Computing*, 27(2):319-340, 1998.

[6] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[7] Garg V.K. and Raynal M., Normality: a Correctness Condition for Concurrent Objects. *Parallel Processing Letters*, 9(1):123-134, 1999.

[8] Herlihy M.P., Wait-Free Synchronization. *ACM TOPLAS*, 13(1):124-149, 1991.

[9] Herlihy M.P. and Shavit N., The topological Structure of Asynchronous Computabiilty. *Journal of the ACM*, 46(6):858-923,, 1999.

[10] Herlihy M.P. and Wing J.M, Linearizability: a Correctness Condition for Concurrent Objects. *ACM TOPLAS*, 12(3):463-492, 1990.

[11] Jayanti P., Wait-Free Computing. *Proc. 9th Int. Workshop on Distributed Algorithms (WDAG'95)*, Springer-Verlag LNCS #972 (J.M. Hélary and M. Raynal Eds), pp. 19-50, 1995.

[12] Jayanti P., Chandra T.D. and Toueg S., Fault-Tolerant Wait-Free Shared Objects. *Journal of the ACM*, 45(3):451-500, 1998.

[13] Jayanti P., Chandra T.D. and Toueg S., The Cost of Graceful Degradation for Omission Failures. *Inf. Processing Letters*, 71:167-172, 1999.

[14] Kim K.H. (Kane), Action Level Fault-Tolerance. In *Advances in Real-Time Systems*, (S.H. Son, Editor), Prentice Hall, pp. 415-434, 1995.

[15] Kim K.H. (Kane), Analysis of Guaranteed Service Time of Distributed Real-time Objects. *Proc. Third Int. IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORCS'00)*, pp. 408-410, St-Malo (France), 2000.

[16] Kopetz H. and Reisinger J., The Non-Blocking Write Protocol NBC: a Solution to a Real-Time Synchronization Problem. *Proc. 14th Int. IEEE Symposium on Real-Time Systems (RTS'93)*, pp. 131-137, 1997.

[17] Lamport L., Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806-811, 1977.

[18] Lamport L., A Fast Mutual Exclusion Algorithm. *ACM TOCS*, 5(1):1-11, 1987.

[19] Le Lann G. On Real-Time and Non Real-Time Distributed Computing. *Proc. 9th Int. Workshop on Distributed Algorithms (WDAG'95)*, Springer-Verlag,LNCS #972 (J.-M. Hélary and M. Raynal Eds), pp. 51-70, 1995.

[20] Loui M.C. and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Advances on Computer Research*, JAI Press, 4:163-183, 987.

[21] Moir M., Fast, Long-Lived Renaming Improved and Simplified. *Science of Computer Programming*, 30:287-308, 1998.

[22] Moir M. and Anderson J.H., Wait-Free Algorithms for Fast, Long-Lived Renaming. *Science of Computer Programming*, 25:1-39, 1995.

[23] Peterson G.L., Concurrent Reading while Writing. *ACM TOPLAS*, 5(1):46-55, 1983.

[24] Raynal M. Real-Time Dependable Decisions in Timed Asynchronous Distributed Systems. In *Proc. 3rd Int. IEEE Workshop on Object-Oriented Real-time Distributed Systems (WORDS 97)*, Newport Beach (CA), pp.283-290, February 1997.

[25] Raynal M., Asynchronous Protocols to Meet Real-Time Constraints: is it Really Sensible? How to proceed? *Proc. First Int. IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORCS'98)*, pp. 290-297, Kyoto, 1998.

[26] Veríssimo P. and Rodrigues L., Distributed Systems for System Architects (Part II: Fault-Tolerance, Part III: Real-Time). *Kluwer Academic Press*, 623 pages, 2000.