Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies

(EXTENDED ABSTRACT)

Tushar Chandra* Vassos Hadzilacos† Prasad Jayanti‡ Sam Toueg§

1 Background and overview

In a shared-memory system, asynchronous processes communicate via typed shared objects, such as registers, test&sets, and queues. The need to implement an object of one type from objects of other types arises often in such systems. Recent research has focussed mostly on *wait-free* implementations. Such an implementation guarantees that every process can complete every operation on the implemented object in a finite number of its own steps, regardless of whether other processes are fast, slow, or have crashed. From now on, we write "implementation" and "implement" as abbreviations for "wait-free implementation" and "wait-free implement", respectively. If an implementation is not wait-free, we will explicitly state so.

It is known that objects of different types vary widely in their ability to support (wait-free) implementations. For example, using test&set objects, one can implement any object that is shared by at most two processes [Her91]. In contrast, using compare&swap objects, one can implement objects of any type that can be shared by any number of processes [Her91]. Thus, it is natural to ask whether object types can be mapped to levels in a hierarchy, where the level of a type reflects its ability to support wait-free implementations. We seek two properties in such a hierarchy: (1) If a type T is at level N, then, for all types T', one can implement an object of type T' shared by N processes, using only registers and objects of type T. This property ensures that weak types are not placed at high levels of the hierarchy. (2) If a type T is at level N, and $\mathcal{S}$ is a set of types all of which are at lower levels than T, then one cannot implement an object of type T shared by N processes, using any number and any combination of objects of types in $\mathcal{S}$. This property guarantees that there are no clever ways of combining objects classified as weak by the hierarchy (i.e., objects whose types are at low levels) to implement objects that are classified as stronger (i.e., at higher levels).

A hierarchy with Property (1) is known as a *wait-free hierarchy*, and a hierarchy with Property (2) is known as a *robust hierarchy* [Jay93]. Our interest lies in a robust wait-free hierarchy, one which has both of the above properties. The existence of such a hierarchy is an open question. We know, however, that if such a hierarchy exists, it must be the hierarchy $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}$ (or a coarsening of it) defined as follows: $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}$ maps a set $\mathcal{S}$ of types to level N if N is the maximum integer so that an N-consensus object can be implemented using only registers and objects belonging to types in $\mathcal{S}$ [Jay93]. (An *N-consensus object* allows each of a maximum of N processes to access it by proposing a value; the object returns the same value to all accesses, where the value returned is the value proposed by some process. This object plays a central role in realising wait-free implementations: Herlihy's *universal construction* shows that using N-consensus objects and registers one can provide a wait-free implementation of *any* object that is shared by at most N processes [Her91].)

From the universal construction of [Her91], it follows that $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}$ is a wait-free hierarchy. Is $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}$ a robust hierarchy? Equivalently, is $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(\{T, T'\}) = \max(\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(T), \mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(T'))$ for arbitrary types T and T'? While the general question remains open, this paper

*IBM T. J. Watson, tushar@watson.ibm.com

†University of Toronto, vassos@db.toronto.edu, partially supported by a grant from the Natural Sciences and Engineering Research Council of Canada

‡Dartmouth College, prasad@cs.dartmouth.edu, supported by Dartmouth grants 2-01361 and 424820.

§Cornell University, sam@cs.cornell.edu, supported by NSF grant CCR-9102231

PODC 94 - 8/94 Los Angeles CA USA

proves that the answer is yes for the restricted case where T is N-consensus (and T' is arbitrary). Thus, if an object type (together with registers) is powerful enough to "boost" N-consensus into $(N+1)$-consensus, then (together with registers) it is already powerful enough to implement $(N+1)$-consensus! This result is the subject of Section 3.

Wait-free implementations provide an extreme degree of fault-tolerance. They assure that even if just one process survives, it will be able to complete its operations. Researchers have also investigated another form of implementations which support a more modest degree of fault-tolerance [DDS87, LAA87]. These guarantee that correct processes will complete their operations, as long as no more than t processes fail, where t is a specified parameter. Such implementations are called *t-resilient*. It is immediate from the definitions that wait-freedom is equivalent to $(N-1)$-resiliency, where N is the number of processes in the system. Using our result that $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(\{T, N\text{-consensus}\}) = \max(\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(T), N)$, we prove an important connection between t-resilient and wait-free implementations of N-consensus. We describe this connection and its significance below. (The proof is given in Section 4.)

Consider the task of devising a t-resilient implementation of an object shared by N processes, as N decreases. As the ratio of correct processes on which the implementation can rely gets smaller, the task seems to become more and more difficult. For example, a t-resilient implementation which works only when a majority of processes are correct, cannot be used when N becomes smaller than $2t+1$. In the limit, when N becomes $t+1$, the task amounts to providing a *wait-free* implementation of the object. Thus, *prima faciae*, it seems that the ability of a set $\mathcal{S}$ of objects to support a t-resilient implementation of an object shared by N processes is greater than the ability of $\mathcal{S}$ to support a wait-free implementation of that object shared by $t+1$ processes. In this paper we show that this is not the case for the N-consensus object. More specifically, we prove that, for any $N > t > 1$, a set of objects can be used for a t-resilient implementation of N-consensus if and only if it can be used for a wait-free implementation of $(t+1)$-consensus.

The "if" direction of this result is not surprising. To give a t-resilient implementation of consensus among N processes, we can choose $t+1$ of the processes, have them run the given wait-free implementation of $(t+1)$-consensus, and write the result into a register. The remaining processes keep reading that register until the result appears there. The "only-if" direction is not obvious, and is useful in obtaining simple proofs of the impossibility of t-resilient implementations. Suppose we want to prove that there is no t-resilient implementation of N-consensus using certain base objects. Since N can be arbitrarily large relative to t, such an impossibility proof can be difficult. Instead, our result allows us to focus on the simpler task of proving the impossibility of a wait-free implementation of $(t+1)$-consensus.[1] To illustrate this, consider the known fact that there is no wait-free implementation of 3-consensus using only queues and registers [Her88]. From this it follows, using our result, that there is no 2-resilient implementation of N-consensus using only queues and registers, for any $N \geq 3$.

To our knowledge, there is only one previous result relating t-resiliency to wait-freedom. Borowski and Gafni showed that any t-resilient solution to the k-set agreement problem among $N > t$ processes *that uses only registers* can be transformed into a wait-free solution to the k-set agreement problem among $t+1$ processes that also uses only registers [BG93]. Their transformation depends on the fact that the original solution employs *only* registers. In contrast, in our result the set of base objects used in the t-resilient or wait-free implementation is arbitrary.

In the final section, we call attention to the significance of an assumption of the traditional model of shared objects. In the traditional model, when a process applies an operation to an object, the response from the object and the resulting state of the object depend only on what the operation is, without regard to the identity of the process invoking the operation. For example, a *deq* operation on a queue has the same effect, regardless of who invokes it. Consider an alternative model of shared objects in which this is not necessarily the case. More specifically, suppose that the response to an operation and the resulting object state may depend not only on the operation but also on the identity of the process invoking the operation. We prove that the hierarchy $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}$ is *not* robust in this model. The proof exhibits a type *booster* with two properties: $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(\text{booster}) = 1$ and $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(\{\text{booster, 2-consensus}\}) = 3$. Thus, although neither booster objects nor 2-consensus objects can implement 3-consensus, the two types of objects, when used together, can implement 3-consensus. It follows that $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}$ is not robust.

Can we hope to discover a type *booster* with the above two properties in the traditional model? The answer is no: our earlier result that $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(\{T, N\text{-consensus}\}) = \max(\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(T), N)$ rules out the existence of such a type in the traditional model.

[1] The difference in difficulty between these tasks can be appreciated by comparing the proof that there is no wait-free implementation of a 3-consensus object using only registers and 1-bit read-modify-write objects to the proof that there is no 2-resilient implementation of an N-consensus object using only registers and 1-bit read-modify-write objects, for any $N > 2$ [LAA87]. The latter proof is much longer (three pages versus one page) and the arguments are more involved [LAA87].

2 Traditional model

A concurrent system consists of asynchronous processes and shared objects. Each (shared) object has a *type* which characterizes the behavior of the object. A type T is a tuple $(OP, S, \mathcal{P})$, where

- OP is the *set of operations* that may be applied to an object of type T.
- S is the *sequential specification*. It specifies the legal state transitions and responses of the object when operations are applied one after the other, without overlap.
- $\mathcal{P}$ is the *set of virtual process names*. For all $P \in \mathcal{P}$ and $op \in OP$, an object $\mathcal{O}$ of type T is required to support *access procedures* $\texttt{Apply}(P, op, \mathcal{O})$.

Let $T = (OP, S, \mathcal{P})$ be a type, $op \in OP$, and $\mathcal{O}$ be an object of type T. A process invokes the operation op on object $\mathcal{O}$ and obtains a response res by executing $res := \texttt{Apply}(P, op, \mathcal{O})$. Note that P is not necessarily the identity of the process applying the operation; it just needs to be a virtual name in $\mathcal{P}$.

We require that, for all $P \in \mathcal{P}$, there be no more than one instance of $\texttt{Apply}(P, *, \mathcal{O})$ in execution at any time. Since $\texttt{Apply}(P, *, \mathcal{O})$ and $\texttt{Apply}(Q, *, \mathcal{O})$ (for $P \neq Q$) may be executed concurrently, the sequential specification, by itself, is not adequate to characterize $\mathcal{O}$'s behavior. We use *linearizability*, together with the sequential specification, to resolve the behavior of an object in the presence of such concurrent accesses. Informally, linearizability requires that each execution of an access procedure appear to take effect instantaneously, at some point in time between the call and the completion of the access procedure.

Two types, $\texttt{cons}(P_1, P_2, \ldots, P_N)$ and $\texttt{register}$, are central to this paper. $\texttt{cons}(P_1, P_2, \ldots, P_N)$ supports the operations *propose 0* and *propose 1*, and has the following sequential specification: if the first operation applied is *propose u*, then every operation, including the first, gets the response u. The set of virtual process names for $\texttt{cons}(P_1, P_2, \ldots, P_N)$ is $\{P_1, P_2, \ldots, P_N\}$. The type $\texttt{register}$ supports the operations *read* and *write u* (u is arbitrary), and has the following sequential specification: *write u* gets the fixed response *ack* and *read* gets the last value written. We leave the set of virtual process names for $\texttt{register}$ unspecified; it can be any finite set.

Let $T = (OP, S, \mathcal{P})$ be a type and $\mathcal{S}$ be a set of types. We say that $\mathcal{S}$ *implements* T (equivalently, T *has an implementation from* $\mathcal{S}$) if there is a function $\mathcal{I}(O_1, O_2, \ldots, O_n)$ such that each O_i $(1 \leq i \leq n)$ is of a type in $\mathcal{S}$ and $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_n)$ is of type T. Intuitively, $\mathcal{I}$ specifies how the access procedures $\texttt{Apply}(P, op, \mathcal{O})$ (for all $P \in \mathcal{P}$ and $op \in OP$) are implemented in terms of the access procedures supported by $O_1, O_2, \ldots, O_n$. The object $\mathcal{O}$ is the ***implemented object*** or ***derived object***, and objects $O_1, O_2, \ldots, O_n$ are the ***base objects***. $\mathcal{I}$ is a ***wait-free implementation*** if $\texttt{Apply}(P, op, \mathcal{O})$ (for all $P \in \mathcal{P}$ and $op \in OP$) is guaranteed to complete in a finite number of steps. We write "implementation" and "implement" as abbreviations for "wait-free implementation" and "wait-free implement", respectively. If any implementation is not wait-free, we will explicitly state so.

Finally, we repeat the definition of the hierarchy $\texttt{h}_\texttt{m}^\texttt{r}$ given in [Jay93]. $\texttt{h}_\texttt{m}^\texttt{r}$ maps a set $\mathcal{S}$ of types to a positive integer or, to ∞, as follows: $\texttt{h}_\texttt{m}^\texttt{r}(\mathcal{S})$ is the maximum integer N such that $\mathcal{S} \cup \{\texttt{register}\}$ implements $\texttt{cons}(P_1, P_2, \ldots, P_N)$. If there is no such maximum, $\texttt{h}_\texttt{m}^\texttt{r}(\mathcal{S}) = \infty$.

3 consensus cannot boost the level of a type in $\texttt{h}_\texttt{m}^\texttt{r}$

We obtain our main results using a lemma that can be informally stated as follows: If we can implement $(N+1)$-consensus from a set $\mathcal{S}$ of types (that includes $\texttt{register}$) and N-consensus, then we can implement N-consensus from $\mathcal{S}$ and $(N-1)$-consensus. Formally, if $\texttt{cons}(P_1, P_2, \ldots, P_{N+1})$ has an implementation from $\mathcal{S} \cup \{\texttt{cons}(P_1, P_2, \ldots, P_N)\}$, then $\texttt{cons}(P_1, P_2, \ldots, P_N)$ has an implementation from $\mathcal{S} \cup \{\texttt{cons}(P_1, P_2, \ldots, P_{N-1})\}$. By applying this lemma repeatedly we can eliminate the use of consensus base objects in the implementation, thereby obtaining an implementation of $(N+1)$-consensus from $\mathcal{S}$. Formally, if $\texttt{cons}(P_1, P_2, \ldots, P_{N+1})$ has an implementation from $\mathcal{S} \cup \{\texttt{cons}(P_1, P_2, \ldots, P_N)\}$, then $\texttt{cons}(P_1, P_2, \ldots, P_{N+1})$ has an implementation from $\mathcal{S}$. This means that if the types in $\mathcal{S}$ are not strong enough to implement $(N+1)$-consensus, then the use of N-consensus cannot "boost" $\mathcal{S}$ to do so. Formally, $\texttt{h}_\texttt{m}^\texttt{r}(\mathcal{S} \cup \{\texttt{cons}(P_1, P_2, \ldots, P_N)\}) = \max(\texttt{h}_\texttt{m}^\texttt{r}(\mathcal{S}), N)$.

3.1 Preliminary lemmas

Lemma 3.1 $\texttt{cons}(P_1, P_2, \ldots, P_N)$ *implements* $\texttt{cons}(Q_1, Q_2, \ldots, Q_N)$.

Proof Intuitively, Q_i disguises itself as P_i. More precisely, an object $\mathcal{O}$ of type $\texttt{cons}(Q_1, Q_2, \ldots, Q_N)$ is implemented using an object O of type $\texttt{cons}(P_1, P_2, \ldots, P_N)$ as follows: $\texttt{Apply}(Q_i$, propose u, $\mathcal{O})$ simply calls (and returns the same value as) $\texttt{Apply}(P_i$, propose u, $O)$. □

O_1, O_2: objects of type $\texttt{cons}(P_1, P_2, \ldots, P_N)$
GP, SP, DEC: registers, initialized to $\perp$

$\texttt{Apply}(P_i$, propose u_i, $\mathcal{O})$ $\quad (1 \le i \le N)$

1. $GP := \texttt{Apply}(P_i$, propose u_i, $O_1)$
2. **if** $SP = \perp$ **then**
3. $\quad vote_i := GP$
4. **else** $vote_i := SP$
5. $DEC := \texttt{Apply}(P_i$, propose $vote_i$, $O_2)$
6. **return** DEC

$\texttt{Apply}(P$, propose u, $\mathcal{O})$

$SP := u$
if $GP = \perp$ **then**
$\quad DEC := u$
else busy loop until $DEC \neq \perp$

return DEC

Figure 1: *GroupSolo*$(P_1, \ldots, P_N; P)$: non wait-free implementation of $\texttt{cons}(P_1, P_2, \ldots, P_N, P)$

The next result, presented in Figure 1, implements a consensus object shared by $P_1, \ldots, P_N$ and P using registers, and consensus objects shared by only $P_1, \ldots, P_N$. This implementation is not wait-free: Process P may block (see the busy loop statement on Line 4). Processes $P_1, \ldots, P_N$, however, will never block. We now informally describe how this implementation works.

Let $\mathcal{O}$ be a derived object of the implementation in Figure 1, and O_1, O_2, GP, SP and DEC be its base objects. (GP and SP are acronyms for "Group's Proposal" and "Solo Proposal", respectively. DEC stands for "Decision".) Of the $N+1$ processes that may share $\mathcal{O}$, processes $P_1, \ldots, P_N$ act as one group, while process P acts as a solo outsider. Processes in the group reach consensus on their initial proposals by accessing O_1. Each process P_i in the group regards the response of O_1 as the group's proposal to consensus with Process P, and writes it in the register GP (see Line 1). P_i then reads SP to check if the solo process P has published its proposal yet. If SP is blank, P_i attempts to promote the group's proposal as the consensus value between Process P and the group. Otherwise, P_i attempts to promote the value in SP (which is the proposal of the solo process P) as the consensus value. Lines 2, 3 and 4 in which P_i sets the local variable $vote_i$ to either GP or SP implement this strategy. It is possible, however, that some processes in the group find SP blank and consequently promote the group's proposal, while the other processes find a non-blank value in SP that they promote. To reconcile such differences, processes in the group reach consensus on their votes by accessing O_2. The response of O_2 is regarded as the final outcome of consensus between Process P and the group.

The solo process P, on the other hand, begins by publishing its proposal in the register SP. It then reads GP, the register where the group's proposal is published. If GP is blank, P concludes that it is ahead of all the processes in the group and that processes in the group will vote for its (P's) proposal. Thus, P regards its proposal as the outcome of its consensus with the group. On the other hand, if P finds GP non-blank, then P is uncertain of the views of the processes in the group. P therefore blocks itself until the consensus value is published in the register DEC by (some process in) the group.

Lemma 3.2 *Figure 1 presents an implementation of* $\texttt{cons}(P_1, P_2, \ldots, P_N, P)$ *from* $\{\texttt{cons}(P_1, P_2, \ldots, P_N), \texttt{register}\}$. *The implementation is not wait-free: P may block. However, for all* $1 \le i \le N$, P_i *does not block.*

We use this implementation later as a building block. In the rest of the paper, we will refer to it as *GroupSolo*$(P_1, \ldots, P_N; P)$.

Lemma 3.3 *Figure 2 presents an implementation of* $\texttt{cons}(P_1, \ldots, P_N, P, P')$ *from* $\{\texttt{cons}(P_1, \ldots, P_N, P), \texttt{cons}(P_1, \ldots, P_N, P'), \texttt{register}\}$. *The implementation works correctly if* P *and* P' *do not access it concurrently.*

This implementation is also used later as a building block. In the rest of the paper, we will refer to it as *NonConcurrent*$(P_1, \ldots, P_N; P, P')$.

Let Cons_N^n denote the set of types $\{\texttt{cons}(P_{i_1}, P_{i_2}, \ldots, P_{i_n}) | 1 \le i_1 < i_2 < \cdots < i_n \le N\}$.

Lemma 3.4 *Let* $m < N$ *and* $\mathcal{S}$ *be any set of types that includes* $\texttt{register}$. *If* $\texttt{cons}(P_1, P_2, \ldots, P_N)$ *has an implementation from* $\mathcal{S} \cup \mathit{Cons}_N^m$, *then it has an implementation* $\mathcal{I}$ *from* $\mathcal{S} \cup \mathit{Cons}_N^m$ *with the following property: If* $\mathcal{O}$ *is a derived object of* $\mathcal{I}$ *and* O *is a base object of type* $\texttt{cons}(P_{i_1}, P_{i_2}, \ldots, P_{i_m})$, *the*

O: object of type $\mathbf{cons}(P_1, \ldots, P_N, P)$
O': object of type $\mathbf{cons}(P_1, \ldots, P_N, P')$
DEC: register, initialized to $\perp$

$\underline{\mathtt{Apply}(P_i,\ \text{propose}\ u_i,\ \mathcal{O})}$ $(1 \leq i \leq N)$

$v_i := \mathtt{Apply}(P_i,\ \text{propose}\ u_i,\ O)$
$DEC := \mathtt{Apply}(P_i,\ \text{propose}\ v_i,\ O')$
return DEC

$\underline{\mathtt{Apply}(P,\ \text{propose}\ u,\ \mathcal{O})}$

if $DEC = \perp$ **then**
  $DEC := \mathtt{Apply}(P,\ \text{propose}\ u,\ O)$
return DEC

$\underline{\mathtt{Apply}(P',\ \text{propose}\ u',\ \mathcal{O})}$

if $DEC = \perp$ **then**
  $DEC := \mathtt{Apply}(P',\ \text{propose}\ u',\ O')$
return DEC

Figure 2: $NonConcurrent(P_1, \ldots, P_N; P, P')$: restricted implementation of $\mathbf{cons}(P_1, \ldots, P_N, P, P')$

procedure $\mathtt{Apply}(P_i, *, \mathcal{O})$ *does not contain a call to* $\mathtt{Apply}(P_j, *, \mathcal{O})$ *for* $j \neq i$.

When P_i proposes a value to a derived object $\mathcal{O}$ of implementation $\mathcal{I}$, P_i accesses the base objects of $\mathcal{O}$ in order to compute $\mathcal{O}$'s response to its proposal. The above lemma states that P_i does not need to fake the identity of another process while accessing any base object of type $\mathbf{cons}(P_{i_1}, P_{i_2}, \ldots, P_{i_m})$. The proof of this lemma is non-trivial, but is omitted due to space constraints.

3.2 The reduction

Let $\mathcal{S}$ be any set of types that includes **register** and $N \geq 2$. We show how to transform an implementation $\mathcal{I}$ of $\mathbf{cons}(P_1, P_2, \ldots, P_{N+1})$ from $\mathcal{S} \cup \mathcal{C}ons^N_{N+1}$ into an implementation $\mathcal{J}$ of $\mathbf{cons}(Q_1, Q_2, \ldots, Q_N)$ from $\mathcal{S} \cup \mathcal{C}ons^{N-1}_{N+1}$. Informally, if we can implement a consensus object shared by $N+1$ processes using objects of types in $\mathcal{S}$ and consensus objects shared by any N processes, then we can implement a consensus object shared by N processes using objects of types in $\mathcal{S}$ and consensus objects shared by *only* $N-1$ processes. In the following, we give an informal account of how this reduction is developed.

We find it convenient to partition $\mathcal{C}ons^N_{N+1}$ into three disjoint sets $\mathcal{C}$, $\mathcal{D}$, and $\mathcal{E}$, where $\mathcal{C} = \{\mathbf{cons}(P_{i_1}, \ldots, P_{i_{N-2}}, P_N, P_{N+1}) \mid 1 \leq i_1 < i_2 < \cdots < i_{N-2} \leq N-1\}$, $\mathcal{D} = \{\mathbf{cons}(P_1, \ldots, P_{N-1}, P_N)\}$, and $\mathcal{E} = \{\mathbf{cons}(P_1, \ldots, P_{N-1}, P_{N+1})\}$. Thus, $\mathcal{I}$ is an implementation of $\mathbf{cons}(P_1, P_2, \ldots, P_{N+1})$ from $\mathcal{S} \cup \mathcal{C} \cup \mathcal{D} \cup \mathcal{E}$. Let $\mathcal{O}$ be a derived object of $\mathcal{I}$ and the

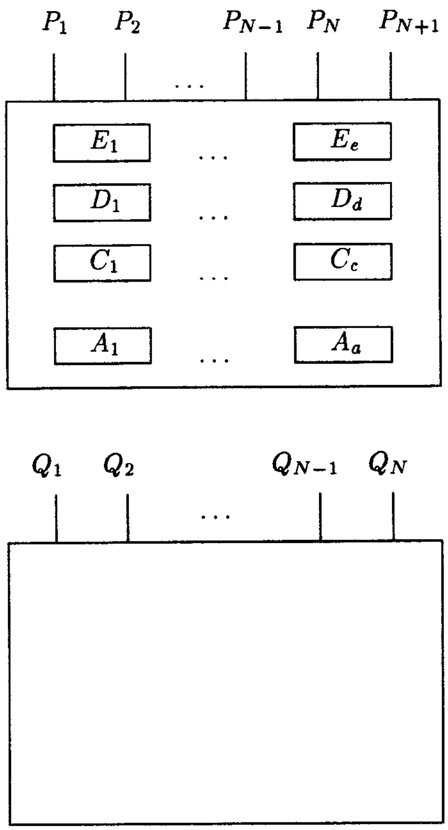

Figure 3: Top: implementation of $\mathcal{O}$ (given); Bottom: implementation of $\mathcal{O}'$ (to be developed)

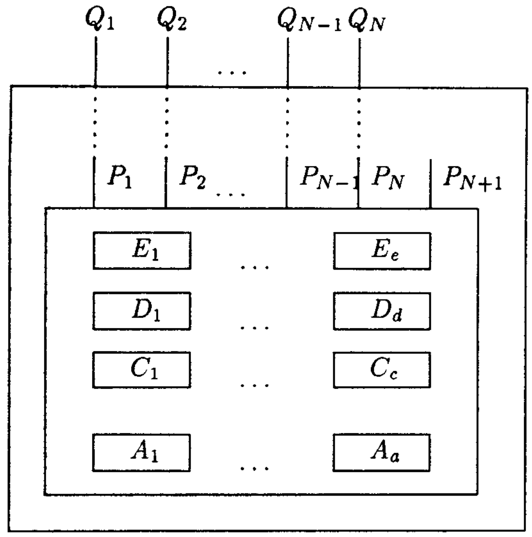

Figure 4: Implementing $\mathcal{O}'$: the first idea

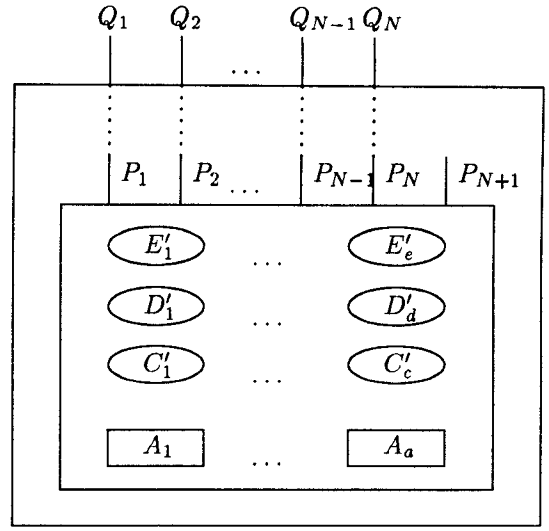

Figure 5: Implementing $\mathcal{O}'$: the second refinement

following be its base objects: $A_1, \ldots, A_a$ of types in $\mathcal{S}$, consensus objects $C_1, \ldots, C_c$ of types in $\mathcal{C}$, consensus objects $D_1, \ldots, D_d$ of the type in $\mathcal{D}$, and consensus objects $E_1, \ldots, E_e$ of the type in $\mathcal{E}$. The first picture in Figure 3 depicts this implementation. By Lemma 3.4, we can assume that $\texttt{Apply}(P_i, *, \mathcal{O})$ does not contain a call to $\texttt{Apply}(P_j, *, O)$ for any $j \neq i$ and $O \in \mathcal{C} \cup \mathcal{D} \cup \mathcal{E}$.

As already mentioned, our goal is to realize $\mathcal{J}$ which implements a consensus object $\mathcal{O}'$ that can be shared by $Q_1, \ldots, Q_N$. Essentially, we must complete the currently blank picture in Figure 3. As we do this, we must bear in mind that base consensus objects of $\mathcal{O}'$ may not be shared by more than $N-1$ processes. The intuition behind how we implement $\mathcal{O}'$ is explained below in a series of steps. Each step proposes an implementation of $\mathcal{O}'$. The implementation may be deficient in certain ways, but these deficiencies will be corrected in the next step (which may introduce new deficiencies).

1. Implement $\mathcal{O}'$ as in Figure 4. That is, implement $\mathcal{O}$ (using $\mathcal{I}$) from the base objects $A_1, \ldots, A_a, C_1, \ldots, C_c, D_1, \ldots, D_d, E_1, \ldots, E_e$, and let $\texttt{Apply}(Q_i$, propose u_i, $\mathcal{O}')$ simply call $\texttt{Apply}(P_i$, propose u_i, $\mathcal{O})$. (Hereafter we will write "Q_i simulates P_i" to mean "$\texttt{Apply}(Q_i$, propose u_i, $\mathcal{O}')$ calls $\texttt{Apply}(P_i$, propose u_i, $\mathcal{O})$".)

   This implementation of $\mathcal{O}'$ is not acceptable: the base objects $D_1, D_2, \cdots, D_d$ belong to the type in $\mathcal{D}$ and are thus shared by N processes—$P_1, P_2, \ldots, P_N$. This violates the requirement that base consensus objects in $\mathcal{O}'$ may be shared by at most $N-1$ processes.

2. Modify the above implementation as follows. For all $1 \leq i \leq c$, replace C_i (of type $\texttt{cons}(P_{i_1}, \ldots, P_{i_{N-2}}, P_N, P_{N+1})$) by C'_i, where C'_i is implemented using the *NonConcurrent*$(P_{i_1}, \ldots, P_{i_{N-2}}; P_N, P_{N+1})$ implementation described in Figure 2. For all $1 \leq i \leq d$, replace D_i by D'_i, where D'_i is implemented using the *GroupSolo*$(P_1, \ldots, P_{N-1}; P_N)$ implementation described in Figure 1. For all $1 \leq i \leq e$, replace E_i by E'_i, where E'_i is implemented using the *GroupSolo*$(P_1, \ldots, P_{N-1}; P_{N+1})$ implementation described in Figure 1. The resulting implementation is depicted in Figure 5.

   This implementation satisfies the requirement that base consensus objects are shared by at most $N-1$ processes, but it has the following deficiency. As Q_N simulates P_N, P_N may apply an operation to an object D'_i and execute the procedure $\texttt{Apply}(P_N, *, D'_i)$. Due to the nature of the *GroupSolo*$(P_1, \ldots, P_{N-1}; P_N)$ implementation with which we implemented D'_i, P_N may enter a busy loop and block while executing $\texttt{Apply}(P_N, *, D'_i)$. (Recall that this happens if P_N finds the base register GP of D'_i to be non-blank, but base register DEC to be blank.) Thus, $\mathcal{O}'$ is not wait-free: $\texttt{Apply}(Q_N, *, \mathcal{O}')$ may block. This is the only problem with this implementation; in particular, $Q_1, \ldots, Q_{N-1}$ do not block.

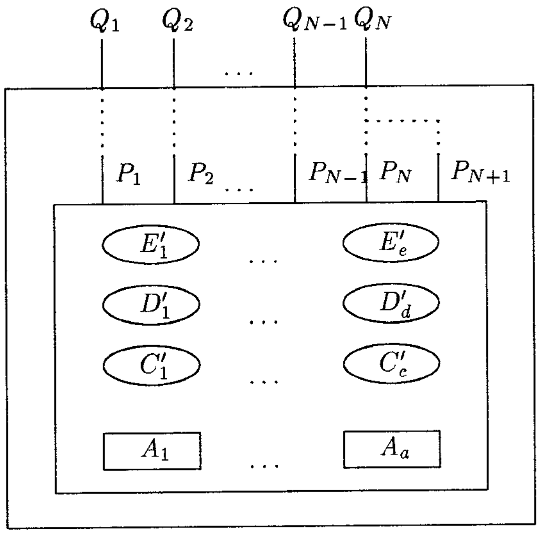

Figure 6: Implementing $\mathcal{O}'$: the third refinement

3. Modify the above implementation as follows. As before, Q_N begins by simulating P_N. However, if P_N enters a busy loop whose termination condition does not hold, Q_N suspends the simulation of P_N and begins simulating P_{N+1}. If P_{N+1} enters a busy loop whose termination condition does not hold, Q_N suspends the simulation of P_{N+1} and resumes the simulation of P_N; and so on. Thus, Q_N simulates P_N and P_{N+1}, switching from one to the other only if the current simulation is stuck in a busy loop. Q_N completes its operation on $\mathcal{O}'$ if (its simulation of) either P_N or P_{N+1} completes. More precisely, the procedure **Apply**(Q_N, propose $u, \mathcal{O}'$) completes as soon as one of the simulated procedures, **Apply**(P_N, propose $u, \mathcal{O}$) or **Apply**(P_{N+1}, propose $u, \mathcal{O}$), completes. Furthermore, **Apply**(Q_N, propose $u, \mathcal{O}'$) returns the same value as the simulated procedure that completes. Figure 6 depicts this implementation.

   If $N \geq 3$, the above strategy is not sufficient to make $\mathcal{O}'$ wait-free: **Apply**(Q_N, propose $u, \mathcal{O}'$) may still block since simulations of P_N and P_{N+1} may *both* block, as illustrated by the following scenario. Processes $Q_3, \ldots, Q_{N-1}$ crash from the beginning. Q_1 and Q_2 crash while simulating accesses of P_1 and P_2, respectively, to two distinct consensus objects implemented by GroupSolo (i.e., D'_i or E'_j objects). Specifically, each crashes after writing the corresponding GP base register, but before writing the corresponding DEC base register. This is precisely the state that may cause the "solo process" in the GroupSolo implementation to block. It is possible that P_N and P_{N+1} are the solo processes in these two GroupSolo implementations. Thus, both the P_N and P_{N+1} threads that Q_N is simulating may become blocked.

4. We overcome the above deficiency by ensuring that P_N and P_{N+1} are not both stuck at the same time. We achieve this by preserving the following property *at all times*: Of the objects in $\{D'_1, \ldots, D'_d, E'_1, \ldots, E'_e\}$, there is no more than one object whose GP register is non-blank and DEC register is blank. To realize this property, we enforce the following discipline on the group of processes $P_1, P_2, \ldots, P_{N-1}$.

   When P_i $(1 \leq i \leq N-1)$ wants to apply an operation *propose* u_i to an object in $\{D'_1, \ldots, D'_d, E'_1, \ldots, E'_e\}$, P_i will be allowed to do so only after it completes all *propose* operations that have already been initiated by processes in the group $\{P_1, P_2, \ldots, P_{N-1}\}$ on objects in $\{D'_1, \ldots, D'_d, E'_1, \ldots, E'_e\}$. To implement this discipline, we use additional (multi-valued) consensus objects — $O_1, \ldots, O_{d+e}$ — that are shared by $P_1, P_2, \ldots, P_{N-1}$. When P_i wishes to propose u to an object $F \in \{D'_1, \ldots, D'_d, E'_1, \ldots, E'_e\}$, P_i must first "obtain permission" to do so. To obtain the permission, P_i accesses $O_1, \ldots, O_{d+e}$, in that order, as described below. P_i proposes the tuple $\langle F, u\rangle$ to O_1. Let $\langle G, v\rangle$ be the response. There are two cases: either $G = F$ or $G \neq F$. (clearly, if $G \neq F$, it means that some process has already obtained permission from O_1 to propose v to object $G \in \{D'_1, \ldots, D'_d, E'_1, \ldots, E'_e\}$.) In the former case, where F and G are the same, P_i proposes v to object G by executing *res* := **Apply**(P_i, propose v, G). P_i then considers its operation of proposing u to F as having completed: the response of F to its operation is *res*. In the latter case, where G is different from F, P_i proposes v to object G by executing *res* := **Apply**(P_i, propose v, G). P_i then accesses O_2 for permission to propose u to F. It does this by proposing $\langle F, u\rangle$ to O_2, and proceeds as above.

   The above discipline ensures that if $F_1, F_2, \ldots$ are the objects in $\{D'_1, \ldots, D'_d, E'_1, \ldots, E'_e\}$ for which $O_1, O_2, \ldots$ give permissions, then the base register DEC of F_i is written before the base register GP of F_{i+1} is written. Thus, at any time, there is at most one object in $\{D'_1, \ldots, D'_d, E'_1, \ldots, E'_e\}$ whose GP register is non-blank and DEC register is blank. This guarantees that, as Q_N simulates P_N and P_{N+1}, P_N and P_{N+1} cannot both block simultaneously. Thus, Q_N will be able to sim-

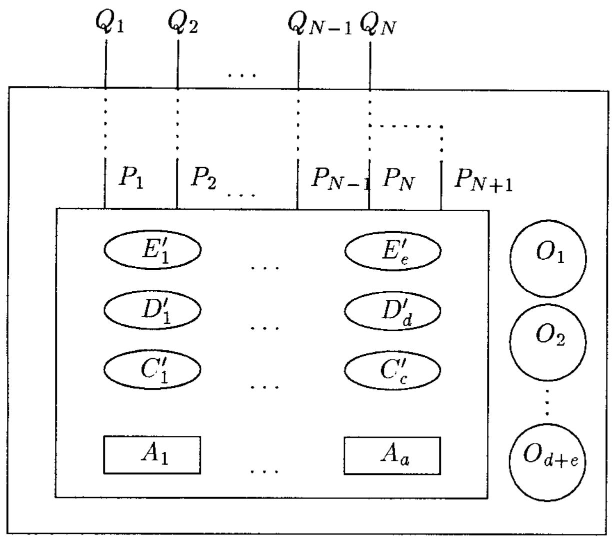

Figure 7: Implementing $\mathcal{O}'$: the fourth and the final refinement

ulate one of P_N and P_{N+1} to completion. Thus, finally, we have the implementation $\mathcal{J}$ of $\mathcal{O}'$ which satisfies all requirements, including wait-freedom. This implementation is depicted in Figure 7.

The following lemma states what the reduction, informally described above, achieves.

Lemma 3.5 *Let $N \geq 2$ and $\mathcal{S}$ be any set of types that includes* `register`. *If* $\mathtt{cons}(P_1, P_2, \ldots, P_{N+1})$ *has an implementation from* $\mathcal{S} \cup Cons^N_{N+1}$, *then* $\mathtt{cons}(Q_1, Q_2, \ldots, Q_N)$ *has an implementation from* $\mathcal{S} \cup Cons^{N-1}_{N+1}$.

By Lemma 3.1, $\mathtt{cons}(P_1, P_2, \ldots, P_{N-1})$ implements each type in $\mathcal{C}\mathrm{ons}^{N-1}_{N+1}$. Thus, we obtain the following corollary of Lemma 3.5.

Corollary 3.1 *Let $N \geq 2$ and $\mathcal{S}$ be any set of types that includes* `register`. *If* $\mathtt{cons}(P_1, P_2, \ldots, P_{N+1})$ *has implementation from* $\mathcal{S} \cup \{\mathtt{cons}(P_1, P_2, \ldots, P_N)\}$, *then* $\mathtt{cons}(P_1, P_2, \ldots, P_N)$ *has implementation from* $\mathcal{S} \cup \{\mathtt{cons}(P_1, P_2, \ldots, P_{N-1})\}$.

3.3 The main theorems

Theorem 3.1 *Let $N \geq 1$ and $\mathcal{S}$ be any set of types that includes* `register`. *If* $\mathtt{cons}(P_1, P_2, \ldots, P_{N+1})$ *has implementation from* $\mathcal{S} \cup \{\mathtt{cons}(P_1, P_2, \ldots, P_N)\}$, *then* $\mathtt{cons}(P_1, P_2, \ldots, P_{N+1})$ *has implementation from* $\mathcal{S}$.

Proof Suppose $\mathtt{cons}(P_1, P_2, \ldots, P_{N+1})$ has an implementation from $\mathcal{S} \cup \{\mathtt{cons}(P_1, P_2, \ldots, P_N)\}$. By repeated application of Corollary 3.1, it follows that $\mathtt{cons}(P_1, P_2, \ldots, P_{N+1})$ has an implementation from $\mathcal{S} \cup \{\mathtt{cons}(P_1)\}$. It is trivial to implement $\mathtt{cons}(P_1)$ from `register`. Hence the theorem. □

Theorem 3.2 *Let $N \geq 1$ and $\mathcal{S}$ be any set of types.* $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathcal{S} \cup \{\mathtt{cons}(P_1, P_2, \ldots, P_N)\}) = \max(\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathcal{S}), N)$

Proof Let $M = \mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathcal{S} \cup \{\mathtt{cons}(P_1, \ldots, P_N)\})$. It is obvious that $M \geq \max(\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathcal{S}), N)$. Suppose, for a contradiction, $M > \max(\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathcal{S}), N)$. By definitions of M and $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$, $\mathtt{cons}(P_1, P_2, \ldots, P_M)$ has an implementation from $\mathcal{S} \cup \{\mathtt{register}, \mathtt{cons}(P_1, \ldots, P_N)\}$. Since $M > N$, $\mathtt{cons}(P_1, P_2, \ldots, P_M)$ has an implementation from $\mathcal{S} \cup \{\mathtt{register}, \mathtt{cons}(P_1, \ldots, P_{M-1})\}$. By Theorem 3.1, $\mathtt{cons}(P_1, P_2, \ldots, P_M)$ has an implementation from $\mathcal{S} \cup \{\mathtt{register}\}$. Thus, by definition of $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$, $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathcal{S}) \geq M$, a contradiction. □

4 Wait-freedom vs. t-resiliency

Theorem 4.1 *Let $\mathcal{S}$ be any set of types that includes* `register` *and N, t be any positive integers such that $N > t \geq 2$.* $\mathtt{cons}(P_1, P_2, \ldots, P_N)$ *has a t-resilient implementation from $\mathcal{S}$ if and only if* $\mathtt{cons}(Q_1, Q_2, \ldots, Q_{t+1})$ *has a wait-free implementation from $\mathcal{S}$.*

Proof Sketch The "if" direction of the theorem is easy. Processes $P_1, \ldots, P_{t+1}$ participate in consensus by simulating $Q_1, \ldots, Q_{t+1}$, respectively. They write the decision value in a register, say, DEC. Processes $P_{t+2}, \ldots, P_N$ simply wait until the decision value is written in DEC and then decide that value.

We now sketch the proof of the "only if" direction. Suppose that $\mathtt{cons}(P_1, P_2, \ldots, P_N)$ has a t-resilient implementation $\mathcal{I}$ from a set $\mathcal{S}$ of types that includes `register`. Using $\mathcal{I}$, we obtain a *wait-free* implementation $\mathcal{I}'$ of $\mathtt{cons}(Q_1, Q_2, \ldots, Q_{t+1})$ from $\mathcal{S} \cup \{\mathtt{test\&set}\}$ as described below. (We will refer to $Q_1, \ldots, Q_{t+1}$ as processes and refer to $P_1, \ldots, P_N$ as threads.) We let the $t+1$ processes — $Q_1, \ldots, Q_{t+1}$ — of implementation $\mathcal{I}'$ simulate the (code associated with the) N threads — $P_1, \ldots, P_N$ — of implementation $\mathcal{I}$, as follows. Process Q_i attempts to simulate all N threads in a fair fashion, say, by executing the first instruction of each thread, and then the second instruction of each thread, and so on. To prevent multiple processes from executing the same instruction of a thread, we require that a process gain exclusive access to a thread before executing an instruction on behalf of that thread. We implement this by associating a test&set object T_j with each thread P_j. When a process Q_i wishes to simulate a thread P_j, it attempts to get exclusive access to P_j by performing the test&set operation on T_j. If Q_i wins T_j (that is, it gets the response 0 from T_j), it observes the current state of P_j, executes the statement pointed to by the program counter of P_j, updates the state of P_j by storing the result of this statement and updating the program counter. Q_i then resets T_j (so that some other process may carry out the next statement of thread P_j) and moves on to thread P_{j+1}. On the other hand, if Q_i loses T_j (that is, it gets the response 1 to its test&set operation on T_j), Q_i simply moves on to thread P_{j+1}. To make the description of our simulation complete, we mention one further detail: we assume, without loss of generality, that the first statement in the code of each thread P_j is to write P_j's proposal in a register R_j private to P_j. In the simulation, the process Q_i that gets to simulate the first statement of P_j writes Q_i's proposal in R_j.

Notice that the crash of a process Q_i will make at most one thread inaccessible for simulation by other processes. Thus, even if up to t processes crash, the remaining process will be able to simulate at least $N - t$ threads in a fair fashion. Since $\mathcal{I}$ is a t-resilient implementation, these threads will run to completion, revealing the consensus value to the correct process. In other words, $\mathcal{I}'$ is a wait-free implementation of $\texttt{cons}(Q_1, Q_2, \ldots, Q_{t+1})$ from $\mathcal{S} \cup \{\texttt{test\&set}\}$. It is known that $\texttt{test\&set}(Q_1, \ldots, Q_{t+1})$ has an implementation from $\{\texttt{cons}(Q_1, Q_2), \texttt{register}\}$ [AGMT92]. Composing this implementation with $\mathcal{I}'$, we conclude that $\texttt{cons}(Q_1, Q_2, \ldots, Q_{t+1})$ has a wait-free implementation from $\mathcal{S} \cup \{\texttt{cons}(Q_1, Q_2)\}$. By Theorem 3.1, this implies that $\texttt{cons}(Q_1, Q_2, \ldots, Q_{t+1})$ has a wait-free implementation from $\mathcal{S}$. □

We remark that the "if" direction of the above theorem holds even for $t = 1$. It is open whether the "only if" direction holds for $t = 1$.

Next we state a universality result for t-resilient implementations. An implementation of a type T is *strongly t-resilient* if every derived object $\mathcal{O}$ has the following property: if a correct process Q calls and executes an access procedure $\texttt{Apply}(P, op, \mathcal{O})$, the procedure will eventually terminate and return a response provided that no more than t processes crash while executing access procedures of $\mathcal{O}$. Notice that the definition takes the crash of a process into account only if the crash occurs while the process is executing an access procedure.

Using similar ideas as in Theorem 4.1, the following can be shown: if $\texttt{cons}(P_1, P_2, \ldots, P_{t+1})$ has a wait-free implementation from a set $\mathcal{S}$ of types (that includes $\texttt{register}$), then $\texttt{cons}(P_1, P_2, \ldots, P_N)$ (for any N) has a strongly t-resilient implementation from $\mathcal{S}$. From this and Herlihy's universal construction [Her91], we obtain:

Theorem 4.2 *Let $t \geq 0$, T be any type, and $\mathcal{S}$ be any set of types that includes* $\texttt{register}$. *If $\texttt{cons}(P_1, P_2, \ldots, P_{t+1})$ has a wait-free implementation from $\mathcal{S}$, then T has a strongly t-resilient implementation from $\mathcal{S}$.*

In the above theorem, an object of type T can be shared by N processes, for any N. This is because $T = (OP, S, \mathcal{P})$ is an arbitrary type and therefore the cardinality N of $\mathcal{P}$ is arbitrary.

5 Robustness in an alternative model

In this section, we call attention to the significance of the following two assumptions of the traditional model of shared objects: (1) The behavior of an object depends only on the operation invoked, without regard to the virtual process name used in the invocation. More specifically, if $\texttt{Apply}(P_i, op, \mathcal{O})$, executed from state σ, returns v and causes $\mathcal{O}$ to move to state σ', then so does $\texttt{Apply}(P_j, op, \mathcal{O})$. (2) There is no restriction on which access procedure a process may use to apply an operation on an object. For example, to apply *op* on $\mathcal{O}$, a process Q may call $\texttt{Apply}(P_i, op, \mathcal{O})$ and later, to apply *op* again, Q may call $\texttt{Apply}(P_j, op, \mathcal{O})$ $(j \neq i)$.

Our reduction in Section 3 depends crucially on the second assumption: recall how Q_N uses $\texttt{Apply}(P_N, *, *)$ sometimes and uses $\texttt{Apply}(P_{N+1}, *, *)$ at other times. This led us to consider an alternative model in which (i) we drop Assumption 1, and (ii) we require that, for each object $\mathcal{O}$ in the system, each process Q in the system be bound to a particular virtual process name P of $\mathcal{O}$. To apply an operation *op* on $\mathcal{O}$, Q may *only* call $\texttt{Apply}(P, op, \mathcal{O})$. Thus, in this alternative model, the response from an object may depend not only on the operation being invoked, but also on the identity of the process invoking the operation.

Interestingly, the results of Section 3 do not hold in this model. Furthermore, as the following theorem states, $\texttt{h}_\texttt{m}^\texttt{r}$ is not robust in this model. The proof is non-trivial, but is omitted due to space constraints.

Theorem 5.1 *In the model described above, there is a type* $\texttt{booster}$ *with the following two properties:* $\texttt{h}_\texttt{m}^\texttt{r}(\texttt{booster}) = 1$ *and* $\texttt{h}_\texttt{m}^\texttt{r}(\{\texttt{booster}, \texttt{cons}(P_1, P_2)\}) = 3$. *Thus,* $\texttt{h}_\texttt{m}^\texttt{r}$ *is not robust in this model.*

One might wonder if a type $\texttt{booster}$ with the above properties also exists in the traditional model of shared objects. If it does, $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$ is not robust even in the traditional model. However, Theorem 3.2 rules out the existence of such a type in the traditional model. Thus, we must look for other (perhaps more complicated) ways if we were to succeed in proving that $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$ is not robust. On the other hand, since robustness of $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$ subsumes Theorem 3.2, we believe that a proof of robustness would also be very involved.

References

[AGMT92] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *Proceedings of the 11th Annual Symposium on Principles of Distributed Computing*, pages 47–58, August 1992.

[BG93] E. Borowsky and E. Gafni. Generalized FLP result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100, 1993.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[Her88] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, 1988.

[Her91] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[Jay93] P. Jayanti. On the robustness of herlihy's hierarchy. In *Proceedings of the 12th Annual Symposium on Principles of Distributed Computing*, August 1993.

[LAA87] M.C. Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.