Robust Wait-Free Hierarchies

PRASAD JAYANTI

Dartmouth College, Hanover, New Hampshire

Abstract. The problem of implementing a shared object of one type from shared objects of other types has been extensively researched. Recent focus has mostly been on *wait-free implementations*, which permit every process to complete its operations on implemented objects, regardless of the speeds of other processes. It is known that shared objects of different types have differing abilities to support wait-free implementations. It is therefore natural to want to arrange types in a hierarchy that reflects their relative abilities to support wait-free implementations. In this paper, we formally define robustness and other desirable properties of hierarchies. Roughly speaking, a hierarchy is robust if each type is "stronger" than any combination of lower level types. We study two specific hierarchies: one, that we call $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$, in which the level of a type is based on the ability of an *unbounded* number of objects of that type, and another hierarchy, that we call $\mathsf{h}_{\mathsf{1}}^{\mathsf{r}}$, in which a type's level is based on the ability of a *fixed* number of objects of that type. We prove that resource bounded hierarchies, such as $\mathsf{h}_{\mathsf{1}}^{\mathsf{r}}$ and its variants, are not robust. We also establish the unique importance of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$: every nontrivial robust hierarchy, if one exists, is necessarily a "coarsening" of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*shared memory*; B.4.3 [**Input/Output and Data Communications**]: Interconnections (subsystems)—*asynchronous/synchronous operation*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processor (MIMD)*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming, synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Asynchronous computing, hierarchy, implementation, MIMD, robustness, shared memory, shared objects, synchronization, wait-freedom

1. *Introduction*

Our study concerns concurrent systems in which processes communicate via shared objects. Processes are *asynchronous*: there are no bounds on their relative

---

A preliminary version of this paper appeared in *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing* (Ithaca, N.Y., Aug. 15–18). ACM, New York, 1993, pp. 145–158.

This work was supported by the National Science Foundations (NSF) grants CCR 91-02231 and CCR 94-10421 and a Dartmouth College Startup grant.

Author's address: 6211 Sudikoff Lab for Computer Science, Dartmouth College, Hanover, NH 03755; e-mail: prasad@cs.dartmouth.edu.

Journal of the ACM, Vol. 44, No. 4, July 1997, pp. 592–614.

speeds. Objects are typed. An object's type specifies the operations that may be invoked and the *sequential behavior*: the legitimate responses corresponding to each sequence of nonoverlapping operations. For example, an object of type `register` supports the operations *read* and *write* v, and has the following sequential behavior: a read operation returns the value written by the latest preceding write operation.

In a concurrent system, operations applied by different processes on the same object may overlap in time. Since the type of an object does not specify the behavior of the object in the presence of such overlapping operations, it is necessary to resort to some additional criterion of correctness. A common criterion, and the one used in this work, is *linearizability* [Herlihy and Wing 1990]. By this criterion, each operation, spanning over an interval of time from the invocation of the operation to its response, must appear to occur at some instant in this interval.

In most systems, simple shared objects, such as registers and test&set objects, are supported in hardware, but more complex objects, such as queues, stacks, and sets, are not. Thus, complex shared objects must be implemented in software. This observation led to extensive research on the "implementation problem", which may be phrased as follows: Given a type T, a positive integer n and a set $\mathcal{S}$ of types, implement an object of type T, that may be shared by up to n processes, using only objects belonging to the types in $\mathcal{S}$. If such an implementation exists, we say that $\mathcal{S}$ *implements* T *for* n *processes.*

Our study is restricted to implementations that are wait-free. An implementation is *wait-free* if every process can complete every operation on the implemented object in a finite number of its own steps, regardless of the execution speeds of the remaining processes. Henceforth, we will use the terms *implementation* and *implement* as shorthands for *wait-free implementation* and *wait-free implement*, respectively.

It turns out that types differ in their ability to support implementations. To make this notion precise, we introduce a definition: A type U is *universal for* n *processes* if, for all types T, $\{U$, `register`$\}$ implements T for n processes [Herlihy 1991]. For example, to state that queue is universal for two processes [Herlihy 1991] simply means that, no matter what type T we pick, it is possible to implement an object of type T, shared by two processes, using only queues and registers.[1] Similarly, stating that `queue` is not universal for three processes [Herlihy 1991] means that there is *some* type T such that there is no implementation of an object of type T, shared by three processes, using only queues and registers. Thus, if a type U is universal for n processes but not for $n + 1$ processes, it means that, using registers and objects of type U, we can implement an object of any type for n processes, but there is at least one type such that we cannot implement an object of this type for $n + 1$ processes. We say that a type U is *universal* if, for all $n \geq 1$, U is universal for n processes.

Clearly, for any type, the maximum number of processes for which it is universal is a measure of its ability to support implementations. By this measure, types do differ in their abilities. For example, `compare&swap` is universal [Herlihy 1991], `test&set` is universal for 2 processes, but not for 3 processes

---

[1] We will use the typewriter font for types. Thus, "`queue`" refers to the type and "queue" refers to an object of this type.

[Loui and Abu-Amara 1987; Herlihy 1991], and `register` is universal for 1 process, but not for 2 processes.[2] In fact, for each positive integer k, there is a type that is universal for k processes, but not for $k + 1$ processes [Jayanti and Toueg 1992]. Our study of classifying types is motivated by these differences in the abilities of types.

We seek to classify types into a hierarchy. A hierarchy assigns a level to each type, where a level is a positive integer or ∞. There are several properties that one intuitively associates with a hierarchy. For instance, one expects that a type is assigned a high level in the hierarchy only if it is "strong". We capture this with a property that requires each type at level n to be universal for n or more processes, and call any hierarchy that satisfies this property a *wait-free hierarchy*. One also expects that each type, at any given level, is "stronger" than any combination of types from lower levels. We formalize this with a property that we call *robustness*. This work investigates the existence of robust wait-free hierarchies.

Herlihy [1991] was the first to propose and study a hierarchy of types. There was however an inconsistency in Herlihy [1991] between the formal definition of the hierarchy and its subsequent interpretation. We identify this ambiguity and investigate the two principal hierarchies that result from different ways of resolving this ambiguity. Roughly speaking, in the first hierarchy, which we call h_1^r, and in its variants, the level of a type is based on the capabilities of a *fixed number* of objects of that type. In contrast, in the second hierarchy, which we call h_m^r, a type's level is based on the capabilities of an *unbounded number* of objects of that type. The main result of this paper is that the hierarchies, such as h_1^r, that are based on resource bounds, are not robust. The basic idea of the proof is as follows: We define a new type called `weak-sticky` and show that h_1^r maps this type to a low level. Thus, the hierarchy h_1^r classifies this type as "weak". But then we show that `weak-sticky` is far from being weak: it is universal.

We also establish the unique importance of the hierarchy h_m^r by showing that every robust wait-free hierarchy (if one exists) is necessarily a "coarsening" of h_m^r. Our work leaves open the question of whether there is a nontrivial robust wait-free hierarchy.

The paper is organized as follows: In Section 2, we describe the model. In Section 3, we state the desirable properties of a hierarchy and define the hierarchies h_1^r and h_m^r. We also show that a robust wait-free hierarchy, if it exists, is necessarily a "coarsening" of h_m^r. In Section 4, we present the main result that h_1^r is not a robust hierarchy. We conclude in Section 5 with a mention of recent advances on the question of whether a robust wait-free hierarchy exists.

2. *Model*

Our system model is similar to the one given by Herlihy [1991]. We still repeat the essential elements of this model here so that we can present a rigorous definition of linearizability, which is needed in some proofs.

2.1. I/O AUTOMATA. We model processes and objects as I/O automata. Our description of I/O automata omits many details. The original work by Lynch and

---

[2] See, for example, Chor et al. [1987], Dolev et al. [1987], Loui and Abu-Amara [1987], and Herlihy [1991].

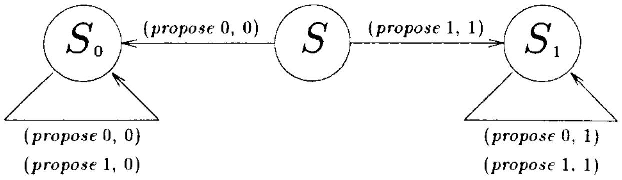

FIG. 1. Sequential specification of consensus.

Tuttle [1988] gives a complete treatment and the work by Herlihy presents the use of I/O automata in modeling shared memory systems [Herlihy 1991].

An I/O automaton (henceforth abbreviated as *automaton*) is described by a set of states, a set of events (partitioned into input, output, and internal events), and a transition relation. An *execution* of an automaton is a sequence s_0, e_1, s_1, e_2, s_2, ... of alternating states and events such that s_0 is a starting state and (s_i, e_{i+1}, s_{i+1}) is a legal transition. A *history* of an automaton is the subsequence of events in an execution. Given "compatible" automata $A_1, A_2, \ldots, A_k$,[3] they can be composed to obtain a new automaton. Let E be an execution of such a composed automaton and H be the corresponding history. The *history of component A_i in E* is the subsequence of H consisting only of the events of A_i.

2.2. TYPE. A *type* is a tuple (OP, RES, Q, δ), where OP is a set of operations, RES is a set of responses, Q is a set of states, and $\delta \subseteq Q \times OP \times Q \times RES$ is a relation, known as the *sequential specification* of the type. Intuitively, if $(\sigma, op, \sigma', res) \in \delta$ it means the following: applying the operation op to an object in state σ can cause the object to move to state σ' and return the response res. δ is required to satisfy two properties:

Totality: For all $\sigma \in Q$ and $op \in OP$, there is at least one pair (σ', res) such that $(\sigma, op, \sigma', res) \in \delta$. (This condition ensures that it is legitimate to apply any operation in any state.)

Computability: There is a computable function $f: Q \times OP \rightarrow Q \times RES$ such that, for all $\sigma \in Q$ and $op \in OP$, $f(\sigma, op) = (\sigma', res)$ implies $(\sigma, op, \sigma', res) \in \delta$. (This condition ensures that a sequential implementation of an object of this type, that is, an implementation that is accessed by only one process, is feasible.)

A type is *deterministic* if, for all $\sigma \in Q$ and $op \in OP$, there is at most one pair (σ', res) such that $(\sigma, op, \sigma', res) \in \delta$. Thus, for deterministic types, δ can be regarded as a function $\delta: Q \times OP \rightarrow Q \times RES$.

A sequence $\Sigma = op_1, res_1, op_2, res_2, \ldots, op_k, res_k$ is *legal from state σ_1 of T* if there are states $\sigma_2, \sigma_3, \ldots, \sigma_{k+1}$ such that, for all i, $1 \leq i \leq k$, $(\sigma_i, op_i, \sigma_{i+1}, res_i) \in \delta$.

The type consensus is central to this paper. For this type, $OP = \{propose\ 0, propose\ 1\}$, $RES = \{0, 1\}$, and $Q = \{S, S_0, S_1\}$. Its sequential specification is given in Figure 1. (In the figure, vertices represent states and there is a directed edge labeled (op, res) from vertex σ to vertex σ' if and only if $(\sigma, op, \sigma', res) \in \delta$.)

---

[3] Automata are *compatible* if they do not share output events and the internal events of each are disjoint from all events of all others.

2.3. OBJECTS, PROCESSES, AND CONCURRENT SYSTEM. Objects and processes are modeled as automata. Each object has three attributes: a name, a type T and an initial value s, which is a state of T. Each process has a name attribute. Below, we define a concurrent system as a composition of processes and objects, where every process can access every object.

A *concurrent system consisting of processes* $P_1, P_2, \ldots, P_n$ *and objects* $O_1, \ldots, O_m$ is defined as the automaton composed from the process automata P_i $(1 \leq i \leq n)$ and the object automata O_j $(1 \leq j \leq m)$. We write $(P_1, P_2, \ldots, P_n; O_1, \ldots, O_m)$ to denote such a concurrent system. For each object O_j of type $T = (OP, RES, Q, \delta)$, its only input events are *invoke*(P_i, op, O_j) and only output events are *respond*(P_i, res, O_j) $(1 \leq i \leq n, op \in OP, res \in RES)$. We call these events *invocations* and *responses*, respectively. For each process P_i, its only input events are *respond*(P_i, res, O_j) and its only output events are *invoke*(P_i, op, O_j).

Let E be an execution of a concurrent system and H be the corresponding history. A *response r matches an invocation i in H* if i is the most recent invocation preceding r such that the process and object names of i and r agree. An *operation execution in H*, abbreviated hereafter as *operation in H*, is a pair of events, an invocation and its matching response.[4] An *incomplete operation* in H is an invocation with no matching response. History H is a *complete history* if it has no incomplete operations.

A precedence relation $<_H$ is defined on the events of H as follows: $e <_H e'$ if and only if event e precedes event e' in H. We abuse notation and extend $<_H$ to also relate "nonoverlapping" operations: For any two operations *oper* and *oper′* in H, *oper* $<_H$ *oper′* if the response of *oper* precedes the invocation of *oper′*. We say that *oper precedes oper′* in H. Two operations unrelated by $<_H$ (i.e., neither operation precedes the other) are said to be *concurrent* in H. History H is *sequential* if it has no concurrent operations.

We assume that a process is a single thread of control: after invoking an operation on an object, it waits to receive the response before it invokes another operation (on any object). We also assume that, for any process P_i and object O_j, the interaction between P_i and O_j is proper: first P_i invokes an operation on O_j, then O_j responds, and then P_i invokes on O_j, then O_j responds, and so on. Formalizing this is straightforward and is omitted.

2.4. LINEARIZABILITY. Linearizability, a correctness criterion for concurrent objects, is due to Herlihy and Wing [1990]. Informally, linearizability requires that each operation, spanning over an interval of time from its invocation to its response, appears to take effect at some instant in this interval.

Let H be the history of some object $\mathbb{O}$ in an execution of a concurrent system. Let $T = (OP, RES, Q, \delta)$ be a type and s be a state of T. A *linearization of H with respect to* (T, s) is a sequence Σ with the following properties:

(1) The elements of Σ are invocations and responses of $\mathbb{O}$. Σ is sequential: each invocation is immediately followed by a matching response. Σ is complete: each invocation has a matching response.

---

[4] Thus, the term, *operation*, is overloaded. It will be however clear from the context whether a particular use of this term refers to an element of *OP* of a type $T = (OP, RES, Q, \delta)$ or to an operation execution in a history.

(2) Σ includes every complete operation in H.

(3) Let invoke(P_i, *op*, $\mathbb{O}$) be an incomplete operation in H (i.e., it has no matching response). Then, either Σ does not include this incomplete operation or Σ includes a complete operation (invoke(P_i, *op*, $\mathbb{O}$), respond(P_i, *res*, $\mathbb{O}$)) (for some $res \in RES$).

Intuitively, this means that an incomplete operation cannot have partial effect: it has either no effect at all or it has full effect.

(4) Σ includes no operations other than the ones mentioned in (2) or (3).

(5) For all operations *oper*, *oper'* in Σ, if $oper <_H oper'$, then $oper <_\Sigma oper'$.

Thus, the order of nonoverlapping operations in H is preserved in Σ.

(6) Σ is legal from state s of T.

Notice that H may have no linearization or may have several different linearizations. We say *H is linearizable with respect to* (T, s) if H has a linearization with respect to (T, s).

We mentioned before that each object has the attributes of type and initial value. The following requirement states what it means to have these attributes.

If $\mathbb{O}$ is an object of type T and initial value s, then each history of $\mathbb{O}$ is linearizable with respect to (T, s).

2.5. IMPLEMENTATION. Our notion of an implementation is similar to Herlihy's [1991] with one exception: Herlihy's definition concerns implementing an object from a single "representation" object, but our definition concerns implementing an object from multiple representation objects. Our definition below is informal, but it can be formalized using the approach in Herlihy [1991].

Let $T_1, T_2, \ldots$ be any finite or infinite sequence of types (a type may appear more than once in the sequence). An *implementation of object $\mathbb{O}$ of type* $T = (OP, RES, Q, \delta)$ *and initial value s from objects $O_1, O_2, \ldots$ of types $T_1, T_2, \ldots$ and initial values $s_1, s_2, \ldots$, respectively, for process names $P_1, P_2, \ldots, P_n$*, consists of a set of procedures `Apply`(P_i, *op*, $\mathbb{O}$) (for each process name P_i, $1 \le i \le n$, and operation $op \in OP$). `Apply`(P_i, *op*, $\mathbb{O}$) specifies how the process, named P_i, should "simulate" the operation *op* on $\mathbb{O}$ in terms of operations on $O_1, O_2, \ldots$; P_i invokes operation *op* on $\mathbb{O}$ by calling `Apply`(P_i, *op*, $\mathbb{O}$). The operation completes when the procedure terminates. The response (from $\mathbb{O}$) to the operation is the value returned by the procedure. The implementation must satisfy the following correctness condition: If $P_1, \ldots, P_n$ are the names of arbitrary process automata, then the history of $\mathbb{O}$, in every execution of $(P_1, P_2, \ldots, P_n; \mathbb{O})$, is linearizable with respect to (T, s). We say $\mathbb{O}$ is the *derived object* and $O_1, O_2, \ldots$ are the *base objects*.[5]

Let T be a type, s be a state of T, and $\mathcal{S}$ be a *set* of types. We say (T, s) *has an implementation from $\mathcal{S}$ for n processes* if there are sequences $T_1, T_2, \ldots$ and $s_1, s_2, \ldots$ such that the following conditions hold:

(1) For all i, T_i is in $\mathcal{S}$ and s_i is a state of T_i.

(2) Given any sequence of objects $O_1, O_2, \ldots$, where O_i's type is T_i and initial value is s_i, it is possible to implement an object of type T and initial value s from $O_1, O_2, \ldots$, for process names $P_1, P_2, \ldots, P_n$.

---

[5] In the terminology of Herlihy [1991], base objects are *representation objects.*

We say *T has an implementation from $\mathcal{S}$ for n processes* if, for each state s of T, (T, s) has an implementation from $\mathcal{S}$ for n processes.

2.6. WAIT-FREE IMPLEMENTATION. *Process P crashes in execution E* if E is an infinite execution and P has only a finite number of events in E; *P is correct in E*, otherwise. An *implementation of object $\mathbb{O}$ for processes $P_1, P_2, \ldots, P_n$ is wait-free* if, in all infinite executions of $(P_1, P_2, \ldots, P_n; \mathbb{O})$ and for all P_i $(1 \leq i \leq n)$, the following is true: If P_i has no incomplete operations on the base objects of $\mathbb{O}$ and if P_i is correct, then P_i has no incomplete operation on $\mathbb{O}$. In this paper, unless qualified otherwise, *implementation* and *implement* stand for *wait-free implementation* and *wait-free implement*, respectively.

2.7. UNIVERSALITY. A *set $\mathcal{S}$ of types is universal for n processes* if every type has an implementation from $\mathcal{S} \cup \{\texttt{register}\}$ for n processes. $\mathcal{S}$ is *universal* if, for all $n > 0$, $\mathcal{S}$ is universal for n processes. If a singleton set $\{T\}$ is universal, we simply say "T is universal".

3. *Classifying Types*

Our objective is to classify types into a hierarchy so that types at higher levels have a greater ability to support implementations than types at lower levels. In this section, we formally state the properties we seek in a hierarchy, define $\mathsf{h}_\mathsf{m}^\mathsf{r}$ and $\mathsf{h}_\mathsf{1}^\mathsf{r}$, the two specific hierarchies investigated in this paper, and present some of their properties.

3.1. DESIRABLE PROPERTIES OF HIERARCHIES. A *hierarchy of types* (henceforth abbreviated as *hierarchy*) is a function that maps types to levels in $\{1, 2, 3, \ldots\} \cup \{\infty\}$. We say that a type T is at level l in hierarchy h if $h(T) = l$. Since hierarchies are just functions, any specific hierarchy is interesting only if it has "useful" properties. We identify such properties below.

P1. If a type is not universal for n processes, then that type is at level $n - 1$ or lower.

P2. If a type is universal for n processes, then that type is at level n or higher.

The first property ensures that a type is not mapped to a higher level than its ability suggests. The second property ensures that it is not mapped to a lower level than its ability suggests. We call a hierarchy that has property P1 a *wait-free hierarchy*, and a hierarchy that has properties P1 and P2 a *tight hierarchy*. Thus, in a tight hierarchy, a type is at level $n < \infty$ if and only if it is universal for n processes, but not for $n + 1$ processes; it is at level ∞ if and only if it is universal. Clearly, there is only one tight hierarchy. In contrast, there are several wait-free hierarchies. For example, a tight hierarchy is a fortiori a wait-free hierarchy. So is the trivial hierarchy which maps every type to level 1.

It is natural to seek a hierarchy in which each type is "stronger" than any combination of lower level types. This motivates the next property:

P3. Let T be a type at level n. Let $m < n$ and $\mathcal{S}$ be a set of types such that each $T' \in \mathcal{S}$ is at level m or lower. Then, T has no implementation from $\mathcal{S}$ for n processes.

A hierarchy that has property P3 is called a *robust hierarchy*. Robustness plays an important role in analyzing the "power" of a *set* of types. We illustrate this with an example. Consider the types `test&set` and `fetch&add`. Each of these is known to be universal for two processes, but not for three processes [Herlihy 1991]. Based *solely* on this knowledge, can we conclude that the set {`test&set`, `fetch&add`} is also not universal for three processes? It is easy to see that the answer depends on whether the (unique) tight hierarchy is robust or not. If it is robust, we can draw such a conclusion. Otherwise, we cannot. More generally, if the tight hierarchy is robust, a set of types is universal for n processes if and only if the set contains a type that is universal for n processes. Thus, the difficult problem of computing the combined power of a set of types reduces to the simpler problem of computing the power of the individual types in the set. On the other hand, if the tight hierarchy is not robust, a set of types could be universal for n processes even if no type in the set is. Thus, it opens up the possibility of implementing a universal type (e.g., `compare&swap`) from a set of nonuniversal types (e.g., {`test&set`, `fetch&add`, . . .}).

3.2. HIERARCHIES $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ AND $\mathsf{h}^{\mathsf{r}}_{\mathsf{1}}$. So far we have identified the desirable properties of a hierarchy, but have not addressed whether there is one with all of these properties. We begin this study by considering the tight hierarchy that we will denote as $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$. Thus, for $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$, robustness is the only one of our three properties open for investigation.

The main drawback of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$, however, is its computability: there appears no easy way of determining the level of a type in $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$. Fortunately, this difficulty is obviated by the following fundamental universality result due to Herlihy [1991].

THEOREM 3.2.1 (HERLIHY'S UNIVERSALITY RESULT). *The type* `consensus` *is universal.*

As an immediate consequence of this result, a type T is universal for n processes if and only if {T, `register`} implements consensus for n processes. This allows us to redefine $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ as follows:

Definition 3.2.2. For each type T, $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}(T)$ is the maximum n such that `consensus` has an implementation from {T, `register`} for n processes. If there is no such maximum, $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}(T)$ is ∞. (Notice that there is no limit on the number of registers or on the number of objects of type T that may be used in the implementation.)

In addition to $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$, in which the level of a type is based on the ability of an unbounded number of objects of that type, one might also consider "resource bounded hierarchies" where a type's level is based on the ability of a fixed number of objects of that type. One such hierarchy, that we call $\mathsf{h}^{\mathsf{r}}_{\mathsf{1}}$ in this paper, is defined below. (The formal definition of the hierarchy in Herlihy [1991] corresponds to $\mathsf{h}^{\mathsf{r}}_{\mathsf{1}}$, but in many results of that paper $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ is used as the hierarchy.)

Definition 3.2.3. For each type T, $\mathsf{h}^{\mathsf{r}}_{\mathsf{1}}(T)$ is the maximum n such that `consensus` has an implementation from {T, `register`} for n processes, where the implementation is restricted to use only one object of type T. (There is no limit on the number of registers that may be used in the implementation.) If there is no such maximum, $\mathsf{h}^{\mathsf{r}}_{\mathsf{1}}(T)$ is ∞.

(In the names of the hierarchies $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$ and $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$, the subscript indicates whether the implementation may use only `1` or many objects of the argument type. The superscript r indicates that the implementation may use registers.)

Is $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ an interesting hierarchy? It is immediate from Herlihy's universality result that $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ is a wait-free hierarchy (i.e., it has property P1). But is $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ tight and/or robust? Is there reason to prefer the definition of $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ over $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$? As we argue below, the answers depend on whether or not $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}} = \mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$. (For any two hierarchies g and h, we say $g = h$ if and only if, for all types T, $g(T) = h(T)$.)

If $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}} = \mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$, computing the level of a type in $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$ becomes simpler. To see this, observe that there are two steps in determining that a type T lies at level n in $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$. First, we must show that a consensus object, shared by n processes, can be implemented using only registers and objects of type T. The second (and perhaps the harder) step is to show that it is impossible to implement a consensus object, shared by $n + 1$ processes, using only registers and objects of type T. If $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}} = \mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$, the second step becomes easier: we will need to show the impossibility only in the case when just a single object of type T is used.

If $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}} \neq \mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$, the hierarchy $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ is neither tight nor robust and is thus hardly interesting. This and some other properties of $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ and $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$ are proved below.

3.3. PROPERTIES OF $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ AND $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$

PROPOSITION 3.3.1. *If h is a wait-free hierarchy, then* $h(\mathtt{register}) = 1$.

PROOF. There exist types (e.g., `queue`) that have no implementation from `register` for two or more processes [Herlihy 1991]. Thus, `register` is not universal for two processes and so, by definition of a wait-free hierarchy, it must be at level less than 2. □

PROPOSITION 3.3.2. $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ *and* $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$ *are both wait-free hierarchies.*

PROOF. Follows trivially from Herlihy's universality result. □

A level k in a hierarchy h is *nonempty* if there is a type T such that $h(T) = k$.

PROPOSITION 3.3.3. *Each level k, $k \in \{1, 2, \ldots\} \cup \{\infty\}$, is nonempty in both* $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ *and* $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$.

PROOF. It was shown in Jayanti and Toueg [1992] that for all $k \in \{1, 2, \ldots\} \cup \{\infty\}$, there is a type T_k such that (i) there is an implementation of `consensus` from T_k for k processes that uses only one object of T_k, and (ii) there is no implementation of consensus from $\{T_k, \mathtt{register}\}$ for $k + 1$ processes. Thus, T_k lies at level k in both $\mathsf{h}_{\mathtt{1}}^{\mathsf{r}}$ and $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$. Hence the proposition. □

Our next result highlights the importance of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$ in the study of robust wait-free hierarchies. Specifically, it states that every robust wait-free hierarchy is a "coarsening" of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$. We begin with the definition of coarsening.

Let $\sigma = (l_1, l_2, \ldots)$ be a finite/infinite sequence such that $l_1 = 1$, $l_1 < l_2 < l_3 \cdots$, and $l_i \in \{1, 2, 3, \ldots\} \cup \{\infty\}$. We say that *hierarchy g is a coarsening of hierarchy h with respect to σ* if, for all types T, we have:

(1) If $l_i \leq h(T) < l_{i+1}$, then $g(T) = l_i$.
(2) If $l_i \leq h(T)$ and l_i is the last element of σ, then $g(T) = l_i$.
(3) If $h(T) = \infty$ and σ is infinite, then $g(T) = \infty$.

Intuitively, levels $l_i, l_i + 1, \ldots, l_{i+1} - 1$ in hierarchy h are lumped into level l_i of hierarchy g, causing levels $l_i + 1 \cdots l_{i+1} - 1$ to be empty in g. The following are some examples.

—Let $\sigma = (1, 3, 5, \ldots)$. Let g be the coarsening of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ with respect to σ. Then, every odd level i of g contains all types that are in levels i and $i + 1$ of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$. All even levels of g are empty. Level ∞ of g contains exactly the same types as level ∞ of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$.

—Let $\sigma = (1, 3)$. Let g be the coarsening of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ with respect to σ. Then, level 1 of g contains the types in levels 1 and 2 of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$, and level 3 of g contains the types in levels 3, 4, ... and level ∞ of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$. Level 2, levels 4, 5, ... and level ∞ of g are empty.

We say that *hierarchy g is a coarsening of hierarchy h* if there is a σ of the form $1 = l_1 < l_2 < l_3 \cdots$ such that g is a coarsening of h with respect to σ. It is easy to verify that (i) every hierarchy is a coarsening of itself, and (ii) if h is a wait-free hierarchy, so is every coarsening of h.

THEOREM 3.3.4. *If h is a robust wait-free hierarchy, then h is a coarsening of* $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$.

PROOF. Assume that h is a robust wait-free hierarchy, and is not a coarsening of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$. Let $\sigma = (l_1, l_2, \ldots)$, where $1 = l_1 < l_2 < l_3 \cdots$ are all the nonempty levels of h. Let g be the coarsening of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ with respect to σ. From our assumption that h is not a coarsening of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$, we have $h \neq g$. Thus, there is a type T such that $h(T) \neq g(T)$. Let $m = g(T)$ and $n = h(T)$. By definition of g, a level k of g is nonempty if and only if level k of h is nonempty. Together with $m \neq n$, this implies that there exist types T' and T'', each different from T, such that $g(T') = n$ and $h(T'') = m$. Since $m \neq n$, we have two cases to consider.

(1) $m > n$. Since g is a coarsening of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ and $g(T) = m$, it follows that $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}(T) \geq m$. Since $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ satisfies Property P1, it follows that T is universal for m processes ($\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ satisfies Properties P1 and P2 because $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ denotes the tight hierarchy). In particular, there is an implementation of T'' from $\{T,$ `register`$\}$ for m processes. Since $h(T) = n < m = h(T'')$, h is not robust. This is a contradiction.

(2) $m < n$. We have the following facts: g is a coarsening of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$, level n of g is nonempty (because $g(T') = n$), $n > m$, and $g(T) = m$. These facts imply $m \leq \mathsf{h}^{\mathsf{r}}_{\mathsf{m}}(T) < n$. Since $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ satisfies Property P2, it follows that T is not universal for n processes. Since $h(T) = n$, it follows that h is not a wait-free hierarchy. This is a contradiction.

This completes the proof of the theorem. □

PROPOSITION 3.3.5. *If $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}} \neq \mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$, then $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}}$ is neither tight nor robust.*

PROOF. Since $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ is the (unique) tight hierarchy, $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}} \neq \mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$ implies $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}}$ is not tight. Further, if $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}} \neq \mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$, it follows from Proposition 3.3.3 that $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}}$ is not a coarsening of $\mathsf{h}^{\mathsf{r}}_{\mathsf{m}}$. Thus, by Theorem 3.3.4 either $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}}$ is not a wait-free hierarchy or $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}}$ is not robust. Since $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}}$ is a wait-free hierarchy (Proposition 3.3.2), it follows that $\mathsf{h}^{\mathsf{r}}_{\mathtt{1}}$ is not robust. □

3.4. STRENGTH OF TYPES. We now remark on our expectation that in a hierarchy each type should be "stronger" than every lower-level type. As we explain below, this expectation holds for $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$, provided that we interpret the phrase "type T_1 is stronger than type T_2" appropriately.

One interpretation of "T_1 is stronger than T_2" is that every problem that can be solved using objects of type T_2 can also be solved using objects of type T_1; furthermore, there is at least one problem that can be solved using objects of type T_1 but cannot be solved using objects of type T_2. We call this the *strong interpretation*. For this interpretation, our expectation that each type should be stronger than every lower-level type does not hold for $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$, as is explained in the next paragraph.

Consider the *2-set agreement problem* among $2n + 1$ processes [Chaudhuri 1990]. This problem can be solved using a single object that has the following behavior: the object remembers the first two values proposed to it and, for each operation, it returns one of these two values nondeterministically as its response [Herlihy and Shavit 1993; Rachman 1994]. The type of this object is at level 1 of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$ [Rachman 1994]. It has also been shown that the 2-set agreement problem among $2n + 1$ processes cannot be solved using only n-consensus objects and registers,[6] despite the fact that the type n-$\mathsf{consensus}$ is at level n of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$.[7] We conclude that, for all $n \geq 2$, there is a problem—the 2-set agreement problem among $2n + 1$ processes—that cannot be solved using objects of a type at level n of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$, but can be solved using objects of a type at level 1 of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$. This implies that, if we use the strong interpretation, our expectation that each type should be stronger than every lower-level type does not hold for $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$.

A second interpretation of "T_1 is stronger than T_2" is that T_1 is universal for a larger number of processes than T_2. We call this the *weak interpretation*. In this interpretation, the strength of a type is associated with the maximum number of processes for which *arbitrary* synchronization tasks are feasible using only objects of that type and registers. For this interpretation, it is immediate from the definition of $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$ that each type is stronger than every lower level type. Furthermore, if $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$ is robust, then each type is stronger than any *set* of types from lower levels. We explain below how a comparison based on this weak interpretation is useful.

Imagine the designer of a multi-processor system who has to decide the type of shared objects that should be supported in hardware. For specificity, suppose that he has to choose between the types T_1 and T_2. Which should he pick? Since the purpose of shared objects is to allow multiple processes to synchronize, the more desirable type is the one that lets a greater number of processes to synchronize. Unfortunately, the designer is not likely to have any knowledge of the kinds of synchronization tasks that potential applications would require processes to engage in. That being the case, the best that the designer can do is to pick the type that maximizes the number of processes among which *arbitrary* synchronization tasks are feasible. In other words, the preferred type is the one that is universal for a larger number of processes. Thus, between types T_1 and T_2, the designer will choose the one that is stronger by the weak interpretation.

---

[6] See, for example, Herlihy and Shavit [1993], Borowsky and Gafni [1993], Saks and Zaharoglou [1993], and Herlihy and Rajsbaum [1994].

[7] The following is an informal specification of n-consensus. To the first n accesses an n-consensus object responds just like a consensus object. The $(n + 1)$st and later accesses get arbitrary nondeterministic responses.

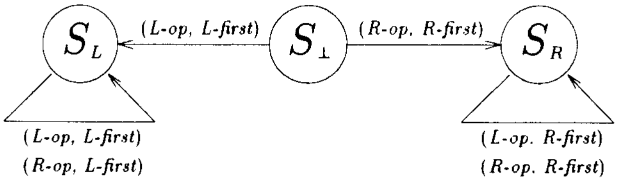

FIG. 2. Sequential specification of the type `sticky`.

Hereafter, when we say that one type is stronger than another, it is always with respect to this weak interpretation.

4. *The Main Theorem*: $\mathsf{h}_1^{\mathsf{r}}$ *is neither Robust nor Tight*

In this section, we prove the main theorem that $\mathsf{h}_1^{\mathsf{r}}$ is neither robust nor tight. We obtain this result by specifying a type `weak-sticky` with the following property: To implement `consensus` from {`weak-sticky`, `register`} for n processes, $n - 1$ weak-sticky objects are both necessary and sufficient. Thus, $\mathsf{h}_1^{\mathsf{r}}$(`weak-sticky`) = 2 and $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$(`weak-sticky`) = ∞. It follows from Proposition 3.3.5 that $\mathsf{h}_1^{\mathsf{r}}$ is neither robust nor tight.[8]

The type `weak-sticky` is specified in Section 4.1. The sufficient and the necessary conditions on the number of weak-sticky objects needed to implement a consensus object are proved in Sections 4.2 and 4.3, respectively.

4.1. SPECIFICATION OF THE TYPE `weak-sticky`.

Consider the type `sticky` in Figure 2. It supports two operations, *L-op* and *R-op*, and responds with either *L-first* or *R-first*. (*L* and *R* stand for Left and Right.) If *L-op* is applied on a sticky object $\mathbb{O}$, initialized to $S_\perp$, $\mathbb{O}$ returns *L-first* as the response. Furthermore, $\mathbb{O}$ returns *L-first* to *all* subsequent operations, reflecting the fact that *L-op* was the first operation applied on $\mathbb{O}$. The behavior is symmetric if, instead of *L-op*, *R-op* was the first operation applied on $\mathbb{O}$. In essence, the first operation "sticks" to $\mathbb{O}$ and determines the response for all operations. `sticky` is similar to the `consensus` [Herlihy 1991] and `sticky-bit` [Plotkin 1989] types.

Now consider the type `weak-sticky`, a variant of `sticky`, shown in Figure 3. Let $\mathbb{O}$ be a weak-sticky object, initialized to $S_\perp$. If *L-op* is the first operation applied on $\mathbb{O}$, $\mathbb{O}$ behaves the same as before. But, `weak-sticky` lacks the symmetry of `sticky`: If *R-op* is the first operation applied on $\mathbb{O}$, *R-op* sticks to $\mathbb{O}$ as before. However, if *R-op* is applied for the second time, it "unsticks" and $\mathbb{O}$ starts behaving as though it had been stuck with *L-op* all along.

The following is an immediate consequence of the definition of `weak-sticky`.

LEMMA 4.1.1. *Let $\mathbb{O}$ be a weak-sticky object, initialized to $S_\perp$. In any execution in which R-op is applied at most once on $\mathbb{O}$, we have*:

(1) *If r_1 and r_2 are the responses to any two operations on $\mathbb{O}$, then $r_1 = r_2$.*

---

[8] A more direct argument that $\mathsf{h}_1^{\mathsf{r}}$ is not robust is as follows. Since $\mathsf{h}_{\mathsf{m}}^{\mathsf{r}}$(`weak-sticky`) = ∞, it follows that every type has an implementation from {`weak-sticky`, `register`} for any number of processes. In particular, even a type mapped to level 3 or higher by $\mathsf{h}_1^{\mathsf{r}}$ has an implementation from {`weak-sticky`, `register`} for any number of processes. This, together with the fact that $\mathsf{h}_1^{\mathsf{r}}$ maps `weak-sticky` to level 2 and `register` to level 1, implies that $\mathsf{h}_1^{\mathsf{r}}$ does not satisfy Property P3 and hence is not robust.

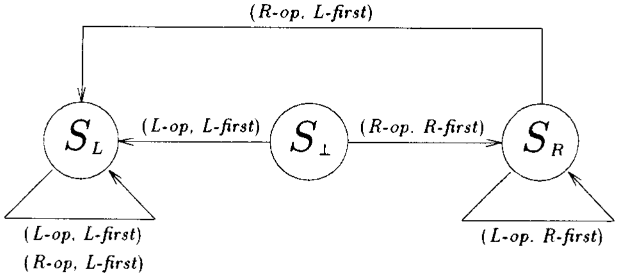

FIG. 3. Sequential specification of the type `weak-sticky`.

(2) *If* $\mathbb{O}$ *returns a response X-first* ($X \in \{L, R\}$), *then an invocation of X-op precedes this response.*

4.2. UPPER BOUND. We show that $n - 1$ weak-sticky objects and (some number of) registers suffice to implement a consensus object for n processes.

We begin by presenting our conventions with respect to implementations of consensus objects.

—As evident from the specification of `consensus` in Figure 1, implementing a consensus object whose initial value is S_0 is trivial: a response of 0 can be returned to every operation. (Initial value of S_1 is similarly trivial.) Thus, the only nontrivial case is to implement a consensus object of initial value $S_\perp$. So when we refer to an implementation as an implementation of consensus, we mean that it is an implementation of (`consensus`, $S_\perp$).

—If $\mathbb{O}$ is a consensus object implemented for processes $P_1, P_2, \ldots, P_n$, then in any execution of $(P_1, P_2, \ldots, P_n; \mathbb{O})$, $\mathbb{O}$ satisfies two properties: (i) $\mathbb{O}$ returns the same response to every invocation, and (ii) $\mathbb{O}$ returns a response v only if *propose* v was already invoked. These are known as the *agreement* and the *validity* properties, respectively. They follow from the specification of `consensus` and the criterion of linearizability.

—By the agreement property of consensus, if a process proposes to a consensus object more than once, the object's responses to the second and subsequent proposals are identical to the object's response to the first proposal by the process. We therefore assume that no process proposes to a consensus object more than once.

We now present our implementation of `consensus`. The implementation is recursive. Let $\mathcal{I}_j$ denote the implementation of `consensus` from {`weak-sticky`, `register`} for processes $P_1, P_2, \ldots, P_j$. The base case is to derive $\mathcal{I}_1$, the implementation of `consensus` for the single process P_1, and is trivial: if $\mathbb{O}_1$ is the derived object, `Apply`(P_1, *propose* v_1, $\mathbb{O}_1$) simply returns v_1. The recursive step of deriving $\mathcal{I}_n$ from $\mathcal{I}_{n-1}$ (for $n \geq 2$) is presented in Figure 4.

LEMMA 4.2.1. *The implementation* $\mathcal{I}_n$ *in Figure* 4 *is a correct implementation of* `consensus` *from* {`weak-sticky`, `register`} *for processes* $P_1, P_2, \ldots, P_n$. $\mathcal{I}_n$ *requires* $n - 1$ *weak-sticky objects and* $2(n - 1)$ *registers.*

PROOF. Notice that $\mathcal{I}_n$ requires one weak-sticky object and two registers in addition to those required by $\mathcal{I}_{n-1}$. Furthermore, $\mathcal{I}_1$ (described above) requires

$\mathcal{O}_{n-1}$: consensus object for $P_1, P_2, \ldots, P_{n-1}$, derived from $\mathcal{I}_{n-1}$
O_{ws}: weak-sticky object, initialized to $S_\perp$
LREG, RREG: binary registers, uninitialized

```
Apply($P_i$, propose $v_i$, $\mathcal{O}_n$) (for $1 \le i \le n-1$)

LREG := Apply($P_i$, propose $v_i$, $\mathcal{O}_{n-1}$)
if Apply($P_i$, L-op, $O_{ws}$) = L-first
    return (LREG)
else return (RREG)
```

```
Apply($P_n$, propose $v_n$, $\mathcal{O}_n$)

RREG := $v_n$
if Apply($P_n$, R-op, $O_{ws}$) = L-first
    return (LREG)
else return (RREG)
```

FIG. 4. Recursive implementation of `consensus` from {`weak-sticky`, `register`} for n processes ($n \ge 2$).

no weak-sticky objects and no registers. This implies that $\mathcal{I}_n$ requires $n - 1$ weak-sticky objects and $2(n - 1)$ registers.

We prove the correctness of $\mathcal{I}_n$ by induction. The following is the induction hypothesis: for $1 \le j \le n - 1$, $\mathcal{I}_j$ is a correct implementation of `consensus` for processes $P_1, P_2, \ldots, P_j$. The base case, namely, that $\mathcal{I}_1$ (described before) is a correct implementation of consensus for P_1, is obvious. The induction step is proved below through several simple claims.

Let $\mathcal{O}_n$ be a derived object of $\mathcal{I}_n$. Consider an execution E of the concurrent system $(P_1, P_2, \ldots, P_n; \mathcal{O}_n)$. Assume that each P_i executes `Apply`(P_i, *propose* v_i, $\mathcal{O}_n$) at most once in E. We make the following claims about E. The proof of each claim follows its statement:

C1. Every process that writes in the register LREG, writes the same value V in LREG. Furthermore, $V \in \{v_1, v_2, \ldots, v_{n-1}\}$.

The claim follows from the agreement and validity properties of $\mathcal{O}_{n-1}$.

C2. No process other than P_n writes in the register RREG. When P_n writes in RREG, it writes the value v_n.

C3. A process receives the response *X-first* from O_{ws} ($X \in \{L, R\}$) only if some process previously completed a write on the register XREG.

By Lemma 4.1.1(2) and the observation that *R-op* is applied at most once on O_{ws}, if a process receives the response *X-first* from O_{ws}, then some process P_k previously invoked *X-op* on O_{ws}. By the implementation, this process P_k completed a write on the register XREG before invoking *X-op*.

Consider the executions of `Apply`(P_i, *propose* v_i, $\mathcal{O}_n$) and `Apply`(P_j, *propose* v_j, $\mathcal{O}_n$) by processes P_i and P_j, respectively. By Lemma 4.1.1(1) and the observation that *R-op* is applied at most once on O_{ws}, the responses received by P_i and P_j from O_{ws} are the same. Let *X-first* be this response (for some $X \in \{L, R\}$). Thus, both P_i and P_j return the value in register XREG. From Claims **C1**, **C2**, and **C3** above, it follows that both P_i and P_j read the same value V in XREG and that $V \in \{v_1, v_2, \ldots, v_n\}$. Thus, the value returned by both P_i and P_j is the same and is from $\{v_1, v_2, \ldots, v_n\}$. We conclude that $\mathcal{O}_n$ satisfies agreement and validity properties. It is obvious that the implementation is wait-free. Hence the correctness of $\mathcal{I}_n$. □

4.3. LOWER BOUND. We prove that any implementation of `consensus` from {`weak-sticky`, `register`} for n processes requires at least $n - 1$ weak-sticky objects, regardless of how many registers it uses. We prove this lower bound in three steps:

(1) We define the notion of *1-trap implementations*. Roughly speaking, an implementation is 1-trap if it is a wait-free implementation for all but at most one correct process. Thus, at most one correct process blocks on such an implementation, and the remaining correct processes complete their operations just as in a wait-free implementation. (The identity of the process that might block is not known a priori.)

(2) We show that if a type T has a 1-trap implementation from `register` for n processes, then any wait-free implementation of consensus from {T, `register`} for n processes requires at least $n - 1$ objects of type T.

(3) We show that `weak-sticky` has a 1-trap implementation from `register`.

4.3.1. k-TRAP IMPLEMENTATIONS. Roughly speaking, an implementation is k-trap if there are at most k processes that, despite taking infinitely many steps, cannot complete their operations on the implemented object. Formally, consider an implementation of an object $\mathbb{O}$ for processes $P_1, P_2, \ldots, P_n$. Let E be an infinite execution of $(P_1, P_2, \ldots, P_n; \mathbb{O})$. We say *a process* P_i *blocks on* $\mathbb{O}$ *in* E if (i) P_i is correct (i.e., P_i has infinitely many events in E), (ii) P_i has no incomplete operations on any of the base objects of $\mathbb{O}$, and (iii) P_i has an incomplete operation on $\mathbb{O}$. An implementation of object $\mathbb{O}$ for processes $P_1, P_2, \ldots, P_n$ is *k-trap* if, for all infinite executions E of $(P_1, P_2, \ldots, P_n; \mathbb{O})$, there are at most k processes that block on $\mathbb{O}$ in E. Notice that (i) a 0-trap implementation is the same as a wait-free implementation, and (ii) in a k-trap implementation, up to k processes may block *unconditionally*—even if there are no process crashes, and even if there are no more than k processes that ever take steps.

4.3.2. A GENERAL LEMMA FOR LOWER BOUNDS. We present a lemma that establishes the utility of k-trap implementations in proving lower-bounds. The proof of this lemma uses the following well-known impossibility result due to Dolev et al. [1987] and Loui and Abu-Amara [1987]. This result is about the *consensus problem for n processes*, defined informally as follows: Each process P_i is initially given an input $v_i \in \{0, 1\}$. Each correct process P_i must eventually decide a value d_i such that (i) $d_i \in \{v_1, v_2, \ldots, v_n\}$, and (ii) for all correct processes P_i and P_j, $d_i = d_j$.

THEOREM 4.3.2.1 [DOLEV ET AL. 1987; LOUI AND ABU-AMARA 1987]. *The consensus problem for n processes has no solution if processes may communicate only via registers and at most one process may crash.*

LEMMA 4.3.2.2. *Let* `T` *be any type such that for every state* σ *of* `T`, *there is a* 1-*trap implementation* $\mathcal{I}_\sigma$ *of* (`T`, σ) *from* `register` *for n processes. Then, any wait-free implementation of consensus from* {`T`, `register`} *for n processes requires at least* $n - 1$ *objects of type* `T`.

PROOF. Suppose that the lemma is false, and there is a wait-free implementation $\mathcal{I}$ of consensus from {`T`, `register`} for n processes such that $\mathcal{I}$ requires

1. For $1 \leq i \leq n-2$, use $\mathcal{I}_{\sigma_i}$ to implement an object O_i of type T initialized to state σ_i. (Thus, each O_i is implemented from just registers.)
2. Use $\mathcal{I}$ to implement a consensus object $\mathcal{O}$ from $O_1, O_2, \ldots, O_{n-2}$ and registers $R_1, R_2, \ldots, R_m$. (Since each O_i is implemented from just registers, it follows that $\mathcal{O}$ is implemented entirely from registers.)
3. Let DECISION be a 3-valued register initialized to $\perp$.
4. For $1 \leq i \leq n$, let v_i be the binary input value of process P_i for the consensus problem. Process P_i executes the procedure `Apply`(P_i, *propose* v_i, $\mathcal{O}$) and writes the return value in register DECISION. As P_i executes this procedure, after each step of the procedure, P_i reads the value in DECISION and if it is not $\perp$, P_i decides this value and terminates.

FIG. 5. 1-resilient consensus protocol $\mathcal{P}$ for n processes.

only $n - 2$ objects of type T, initialized to some states $\sigma_1, \sigma_2, \ldots, \sigma_{n-2}$ of T, and m registers (for some $m \geq 0$). Consider the protocol $\mathcal{P}$ in Figure 5. Clearly, processes communicate exclusively via registers in protocol $\mathcal{P}$. We argue below that $\mathcal{P}$ solves the consensus problem for processes $P_1, P_2, \ldots, P_n$ even if at most one of the processes may crash. By Theorem 4.3.2.1, such a protocol does not exist. Hence the lemma.

We claim that at most $n - 2$ correct processes fail to complete their operations on $\mathbb{O}$. This follows from the following facts:

(1) Object $\mathbb{O}$ is implemented from $O_1, \ldots, O_{n-2}, R_1, \ldots, R_m$. Each O_i is 1-trap: at most one process blocks on it.
(2) Every correct process completes all of its operations on the registers $R_1, R_2, \ldots, R_m$.
(3) The implementation of $\mathbb{O}$ from $O_1, \ldots, O_{n-2}, R_1, \ldots, R_m$ is wait-free. Therefore, if a process P_k is correct and does not block on any of $O_1, \ldots, O_{n-2}$, then P_k will eventually complete executing the procedure `Apply`(P_k, *propose* v_k, $\mathbb{O}$).

Therefore, if at most one of $P_1, P_2, \ldots, P_n$ crashes, there is still one process, call it P_k, that neither crashes nor blocks on $\mathbb{O}$. This process P_k eventually writes the response, call it V, returned by `Apply`(P_k, *propose* v_k, $\mathbb{O}$) in register DECISION. Since $\mathbb{O}$ satisfies validity, we have $V \in \{v_1, v_2, \ldots, v_n\}$. Since $\mathbb{O}$ satisfies agreement, no process ever writes a value different from V in register DECISION. The protocol in Figure 5 ensures that every non-crashing process, even if it blocks on a O_i, eventually reads the register DECISION and decides V. In other words, $\mathcal{P}$ solves the consensus problem for $P_1, P_2, \ldots, P_n$ even if at most a single process may crash. This completes the proof of the lemma. □

4.3.3. 1-TRAP IMPLEMENTATION OF `weak-sticky`. Recall that `weak-sticky` has three states—$S_\perp$, S_L, and S_R. We now present a 1-trap implementation of (`weak-sticky`, $S_\perp$) and 0-trap implementations of (`weak-sticky`, S_L) and (`weak-sticky`, S_R). These implementations will use only registers as base objects.

A 1-trap implementation of (`weak-sticky`, $S_\perp$) from `register` for n processes is presented in Figure 6. This implementation is subtle. It is based on the observation that if the first *R-op* operation is blocked, then all other (*R-op*

$R[1 \dots n]$: binary (1-writer, n-reader) registers initialized to 0

```
Apply(P_i, L-op, O)

return (L-first)
```

```
Apply(P_i, R-op, O)

1. if ($\forall k : R[k] = 0$)
2.     R[i] := 1
3.     repeat until ($\exists j < i : R[j] = 1$)
   endif
4. return (L-first)
```

FIG. 6. 1-trap implementation of (weak-sticky, $S_\perp$) from register.

and L-*op*) operations can legitimately return *L-first*. We present below an informal argument of correctness before giving the formal proof. Consider a weak-sticky object $\mathbb{O}$ implemented as in Figure 6. Let H be a history of $\mathbb{O}$, and let *first-op* denote the first operation to complete in H. There are two cases. Case (1) corresponds to *first-op* being an L-*op* operation. Consider the linearization Σ, which includes only the complete operations in H and sequences them in the order of their completion times. Thus, *first-op*, which is an L-*op* operation, becomes the first operation in Σ. Furthermore, the response of every operation in Σ is *L-first* (since *R-first* is never returned in the implementation). From the sequential specification of `weak-sticky` in Figure 3, it is obvious that Σ is legal from the state $S_\perp$ of `weak-sticky`. Now consider Case (2), which corresponds to *first-op* being an R-*op* operation. The key observation is that if *first-op*, which is an R-*op* operation, completed in H, then by our implementation, there must be another R-*op* operation, call it *blocked-op*, from a different process which is concurrent with *first-op* and is blocked. Let us pretend that, although incomplete, *blocked-op* has indeed taken effect in H and received the response *R-first*. Consider the linearization Σ which sequences *blocked-op* first, *first-op* second, and the remaining complete operations in H in the order of their completion times. (*blocked-op* can be linearized before *first-op* since these two operations are concurrent.) Thus, the first operation in the linearization Σ is an R-*op* operation with *R-first* as the associated response. The second operation in the linearization is also an R-*op* operation, and has *L-first* as the associated response. The remaining operations in the linearization have *L-first* as their response. From the sequential specification of `weak-sticky` in Figure 3, it is obvious that Σ is legal from the state $S_\perp$ of `weak-sticky` and hence is a linearization of H with respect to (`weak-sticky`, $S_\perp$). Hence the correctness of our implementation. We formalize these arguments and present a more rigorous proof of correctness below. The proof is based on a series of claims.

CLAIM 4.3.3.1. *The implementation is* 1-*trap*.

PROOF. Clearly, a correct process P_i blocks if and only if the **repeat until** loop (Statement 3 of `Apply`(P_i, R-*op*, $\mathbb{O}$)) never terminates. By Statement 2, such a P_i finishes writing the value 1 into $R[i]$ before blocking.

Suppose that the claim is false: two correct processes P_i and P_j (assume $j < i$) block on $\mathbb{O}$. It follows that $R[i] = R[j] = 1$ and each of P_i and P_j is in the **repeat until** loop that never terminates. Process P_i eventually reads the value 1 from

$R[j]$ and, since $j < i$, P_i quits the **repeat until** loop and returns *L-first*. This contradicts the assumption that P_i blocks on $\mathbb{O}$. ☐

The next claim asserts that if a process P_i successfully completes an R-*op* operation on $\mathbb{O}$, then a different process P_j is already blocked, unable to complete its R-*op* operation on $\mathbb{O}$.

CLAIM 4.3.3.2. *Let E be an execution of* $(P_1, P_2, \ldots, P_n; \mathbb{O})$, *and H be the corresponding history. Suppose that H contains the two events—an invocation* $e_i^{inv} = inv(P_i, R\text{-}op, \mathbb{O})$ *and its matching response* $e_i^{res} = resp(P_i, L\text{-}first, \mathbb{O})$. *Then H contains an invocation* $e_j^{inv} = inv(P_j, R\text{-}op, \mathbb{O})$ *such that*

(1) $e_j^{inv} <_H e_i^{res}$, *and*
(2) e_j^{inv} *has no matching response in H.*

PROOF. The proof of this claim is based on the following observations:

O1. The predicate $\exists k : R[k] = 1$ is stable: that is, if it holds in some state of an execution, it holds in every subsequent state of that execution. Furthermore, this predicate must hold before a response can occur to any invocation of R-*op*.

The first part of this observation follows from the fact that once a "1" is written to a register, it is never changed. The second part is obvious from Statements (1) and (2) of the implementation.

O2. In H, let k be the smallest integer such that P_k has an invocation $e_k^{inv} = inv(P_k, R\text{-}op, \mathbb{O})$ and P_k writes a 1 in $R[k]$. Then e_k^{inv} has no matching response in H.

To see this, notice that after writing a 1 in $R[k]$, P_k enters the **repeat until** loop. This loop never terminates in H because of our premise that k is the smallest integer such that P_k writes a 1 in $R[k]$. Thus, P_k does not return from `Apply`$(P_k, R\text{-}op, \mathbb{O})$.

O3. In H, if a process P_m writes 1 in $R[m]$ after an invocation $e_m^{inv} = inv(P_m, R\text{-}op, \mathbb{O})$, then $e_m^{inv} <_H e_i^{res}$.

Suppose not. Then $e_i^{res} <_H e_m^{inv}$. After the invocation e_m^{inv}, when P_m executes Statement 1 of the procedure `Apply`$(P_m, R\text{-}op, \mathbb{O})$, the guard $\forall m : R[m] = 0$ evaluates to *false* (by **O1**). Thus, P_m returns the response *L-first* without writing into $R[m]$. This contradicts the premise that P_m writes 1 into $R[m]$ after the invocation e_m^{inv}.

To complete the proof of the claim, let S be the set of processes that invoke R-*op* on $\mathbb{O}$ and write 1 into a register in the execution E. Since H contains a response event e_i^{res}, it follows from **O1** that S is nonempty. Let j be the smallest integer such that $P_j \in S$. By **O2**, P_j's invocation e_j^{inv} of R-*op* on $\mathbb{O}$ has no matching response in H. By **O3**, $e_j^{inv} <_H e_i^{res}$. Hence the claim. ☐

CLAIM 4.3.3.3. *Let E be an execution of* $(P_1, \ldots, P_n; \mathbb{O})$ *and H be the history of* $\mathbb{O}$ *in E. H is linearizable with respect to* (`weak-sticky`, $S_\perp$).

PROOF. If H has no response events, then the claim is trivial: the empty sequence is a linearization of H with respect to (`weak-sticky`, $S_\perp$). Assume, therefore, that H has one or more response events. It is obvious from the implementation that the response of each of these is *L-first*. Let $e_i^{res} = resp(P_i,$

R: binary register, initialized to 0

```
Apply(P_i, L-op, O)

if (R = 0)
    return (R-first)
else return (L-first)
```

```
Apply(P_i, R-op, O)

R := 1
return (L-first)
```

FIG. 7. 0-trap implementation of (weak-sticky, S_R) from register.

L-first, $\mathcal{O}$) be the first response event in H. Let e_i^{inv} be the invocation whose matching response is e_i^{res}. There are two cases:

Case 1. $e_i^{inv} = inv(P_i, L\text{-}op, \mathcal{O})$. This corresponds to the case in which the first operation to complete is an *L-op* operation from process P_i. Define a sequence Σ as follows:

(1) Σ includes all complete operations on $\mathcal{O}$ in H, and no other operation.
(2) If two operations *op* and *op'* are in Σ, then $op <_\Sigma op'$ if and only if response of *op* precedes the response of *op'* in H.

It is easy to verify that Σ is legal from the state $S_\perp$ of `weak-sticky` and that Σ is a linearization of H with respect to (`weak-sticky`, $S_\perp$).

Case 2. $e_i^{inv} = inv(P_i, R\text{-}op, \mathcal{O})$. This corresponds to the case in which the first operation to complete is an *R-op* from process P_i. By Claim 4.3.3.2, there is an invocation $e_j^{inv} = inv(P_j, R\text{-}op, \mathcal{O})$ such that $e_j^{inv} <_H e_i^{res}$ and e_j^{inv} has no matching response in H. Define a sequence Σ as follows:

(1) Σ includes all complete operations on $\mathcal{O}$ in H, the operation $(e_j^{inv}, \mathrm{e}_j^{\mathrm{res}})$, where $e_j^{res} = resp(P_j, R\text{-}first, \mathcal{O})$, and no other operation.
(2) The operation $(e_j^{inv}, \mathrm{e}_j^{\mathrm{res}})$ precedes all other operations in Σ.
(3) If *op* and *op'* are operations in Σ different from $(e_j^{inv}, \mathrm{e}_j^{\mathrm{res}})$, $\mathrm{op} <_\Sigma \mathrm{op}'$ if and only if the response of *op* precedes the response of *op'* in H.
It is easy to verify that Σ is legal from the state $S_\perp$ of `weak-sticky` and that Σ is a linearization of H with respect to (`weak-sticky`, $S_\perp$).

Hence the claim. □

LEMMA 4.3.3.4. *Figure* 6 *presents a* 1-*trap implementation of* (`weak-sticky`, $S_\perp$) *from* `register` *for processes* $P_1, P_2, \ldots, P_n$.

PROOF. Follows from Claims 4.3.3.1 and 4.3.3.3. □

LEMMA 4.3.3.5. *Figure* 7 *presents a* 0-*trap* (*wait-free*) *implementation of* (`weak-sticky`, S_R) *from* `register` *for processes* $P_1, P_2, \ldots, P_n$.

PROOF. Notice that in the implementation, in order to apply *L-op* on $\mathcal{O}$, a process must read register R and, in order to apply *R-op* on $\mathcal{O}$, a process must write register R. Thus, for each operation on $\mathcal{O}$, there is an associated operation on register R.

```
Apply($P_i$, $L$-$op$, $\mathbb{O}$)            Apply($P_i$, $R$-$op$, $\mathbb{O}$)

return ($L$-first)                 return ($L$-first)
```

FIG. 8. 0-trap implementation of (`weak-sticky`, S_L).

Let E be an execution of $(P_1, P_2, \ldots, P_n; \mathbb{O})$, H be the history of $\mathbb{O}$ in E, and H' be the history of R in E. Let Σ' be a linearization of H' with respect to (`register`, 0). We now define a linearization Σ of H. Informally, Σ includes all operations on $\mathbb{O}$ whose associated operations on R took effect; the operations in Σ are sequenced by the order in which their associated operations on R took effect. Formally, the sequence Σ is defined as follows:

(1) Σ includes every complete operation on $\mathbb{O}$ in H.
(2) If invoke(P_i, op, $\mathbb{O}$) is an incomplete operation in H whose associated operation op' on R is in Σ', then Σ includes a complete operation (invoke(P_i, op, $\mathbb{O}$), respond(P_i, res, $\mathbb{O}$)), where res is determined as follows. If op is L-op and op' returned 0, then res is *R-first*. If op is L-op and op' returned 1, then res is *L-first*. If op is R-op, then res is *L-first.*
(3) Σ includes no operation other than the ones mentioned in (1) or (2).
(4) For any two (complete) operations op_1 and op_2 in Σ, op_1 precedes op_2 in Σ if and only if op_1's associated operation on R precedes op_2's associated operation on R in Σ'.

It is easy to verify that Σ is a linearization of H with respect to (`weak-sticky`, S_R). Thus, the implementation is correct. It is obvious that the implementation is wait-free or, equivalently, 0-trap. □

LEMMA 4.3.3.6. *Figure* 8 *presents a* 0-*trap* (*wait-free*) *implementation of* (`weak-sticky`, S_L) *from* `register` *for processes* $P_1, P_2, \ldots, P_n$.

PROOF. Obvious. □

4.3.4. THE LOWER BOUND. The following lower bound is immediate from Lemmas 4.3.2.2, 4.3.3.4, 4.3.3.5, and 4.3.3.6.

LEMMA 4.3.4.1. *Any wait-free implementation of consensus from* {`weak-sticky`, `register`} *for n processes requires at least* $n - 1$ *objects of type* `weak-sticky`.

4.4. THE MAIN THEOREM. By Lemma 4.2.1, h^r_m(`weak-sticky`) $= \infty$ and h^r_1(`weak-sticky`) ≥ 2. By Lemma 4.3.4.1, h^r_1(`weak-sticky`) ≤ 2. Thus, h^r_m(`weak-sticky`) $= \infty$ and h^r_1(`weak-sticky`) $= 2$. From this and Proposition 3.3.5, we have:

THEOREM 4.4.1. h^r_1 *is neither robust nor tight.*

Intuitively, h^r_1 is not robust because it places `weak-sticky` at level 2, below several other types, as if it were a weak type. But `weak-sticky` is far from being weak: it is universal (h^r_m(`weak-sticky`) $= \infty$).

4.5. DISCUSSION. In determining the level of a type in h^r_1, we are restricted to use at most one object of that type. It is this limitation that we exploited in proving that h_1^r is not robust. In fact, if we are restricted to use at most k objects (for any integer k), rather than one object, the resulting hierarchy would still be not robust. The proof again uses `weak-sticky` and is almost identical to the proof of non-robustness of h_1^r.

The definitions of h_1^r and h_m^r allow the use of registers in determining the level of a type. Disallowing the use of registers from h_1^r and h_m^r result in two new hierarchies h_1 and h_m, respectively [Jayanti 1993]. These two hierarchies are also neither robust nor tight [Jayanti 1993]. Kleinberg and Mullainathan [1993] independently prove that h_1 is not robust. Bazzi et al. [1994] prove that for all types T, if either T is deterministic or $\mathsf{h}_m(T) \geq 2$, then $\mathsf{h}_m(T) = \mathsf{h}_m^r(T)$. Thus, for a large class of types, the ability of a type to implement `consensus` is not enhanced by the availability of registers.

5. *Conclusion*

We formally defined robustness and other desirable properties of hierarchies of types. The hierarchy h_1^r was proved to be nonrobust. Its nonrobustness is due to the fact that the level of a type in h_1^r is determined by the ability of a single object of that type. More generally, our results imply that no hierarchy, in which a type's level is determined by the ability of a fixed number of objects of that type, is robust. Thus, our results formally establish that h_m^r, the hierarchy in which the level of a type is based on the ability of an unbounded number of objects of that type, is the only interesting hierarchy. We leave open the question of whether h_m^r is robust.

Robustness of h_m^r plays an important role in analyzing the power of a set of types. If h_m^r is robust, a set of types is universal for n processes if and only if the set contains a type that is universal for n processes. Thus, the difficult problem of computing the combined power of a set of types reduces to the simpler problem of computing the power of the individual types in the set. On the other hand, if h_m^r is not robust, a set of types could be universal for n processes even if no type in the set is. Thus, it opens up the possibility of implementing a universal type from a set of nonuniversal types.

Since the time our results were first published [Jayanti 1993], there have been significant advances on the question of whether h_m^r is robust. Borowsky et al. [1994] and Peterson et al. [1994] prove that, if we only consider deterministic types, then h_m^r is robust. This result is important since most types of interest are deterministic. Using nondeterministic types and with the assumption that a process may not bind itself to more than one "port" of an object, Chandra et al. [1994] prove that h_m^r is not robust. Moran and Rappoport [1996] and Lo and Hadzilacos [1997] strengthen this result in two different ways. Moran and Rappoport prove the nonrobustness of h_m^r without the use of nondeterministic types, but, as in the work of Chandra et al., they assume that a process may not bind itself to more than one "port" of an object [Moran and Rappoport 1996]. Lo and Hadzilacos [1997] prove the nonrobustness of h_m^r without making any assumptions on how processes bind to objects, but their work requires the use of nondeterministic types. Schenk [1997] proves a result similar to the one in Lo and Hadzilacos [1997], but his result assumes "infinite" nondeterminism and applies only for a stronger definition of wait-free implementation. Jayanti [1995] summarizes many of the results in one unified framework.

ACKNOWLEDGMENTS. I thank Jon Kleinberg, Sendhil Mullainathan, and Sam Toueg for extremely helpful discussions. Vassos Hadzilacos, Sanjay Khanna, Sam Toueg, and anonymous referees provided helpful comments on earlier drafts.

REFERENCES

Bazzi, R. A., Neiger, G., and Peterson, G. L. 1994. On the use of registers in achieving wait-free consensus. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, Calif., Aug. 14–17). ACM, New York, pp. 354–363.

Borowsky, E., and Gafni, E. 1993. The implication of the Borowski–Gafni simulation on the set consensus hierarchy. Tech. Rep. 930021. Computer Science Dept., Univ. California at Los Angeles, Los Angeles, Calif.

Borowsky, E., Gafni, E., and Afek, Y. 1994. Consensus power makes (some) sense! In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, Calif., Aug. 14–17). ACM, New York, pp. 363–372.

Chandra, T., Hadzilacos, V., Jayani, P., and Toueg, S. 1994. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, Calif., Aug. 14–17). ACM, New York, pp. 334–343.

Chaudhuri, S. 1990. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Que., Canada, Aug. 22–24). ACM, New York, pp. 311–324.

Chor, B., Israeli, A., and Li, M. 1987. On processor coordination using asynchronous hardware. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 86–97.

Dolev, D., Dwork, C., and Stockmeyer, L. 1987. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (Jan.), 77–97.

Herlihy, M. P. 1991. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst. 13*, 1 (Jan.), 124–149.

Herlihy, M., and Rajsbaum, S. 1994. Set consensus using arbitrary objects. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, Calif., Aug. 14–17). ACM, New York, pp. 324–333.

Herlihy, M., and Shavit, N. 1993. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on the Theory of Computing* (San Diego, Calif., May 16–18). ACM, New York, pp. 111–120.

Herlihy, M. P., and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst. 12*, 3 (July), 463–492.

Jayanti, P. 1993. On the robustness of Herlihy's hierarchy. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing* (Ithaca, N.Y., Aug. 15–18). ACM, New York, pp. 145–158.

Jayanti, P. 1995. Wait-free computing. In *Proceedings of the 9th International Workshop on Distributed Algorithms* (Le Mont-Saint-Michel, France, Sept.). Lecture Notes in Computer Science, vol. 972. Springer-Verlag, New York, pp. 19–50.

Jayanti, P., and Toueg, S. 1992. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th Workshop on Distributed Algorithms* (Haifa, Israel, Nov.). Lecture Notes in Computer Science, vol. 647. Springer-Verlag, New York.

Kleinberg, J. M., and Mullainathan, S. 1993. Resource bounds and combinations of consensus objects. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing* (Ithaca, N.Y., Aug. 15–18). ACM, New York, pp. 133–144.

Lo, W. K., and Hadzilacos, V. 1997. All of us are smarter than any of us: Wait-free hierarchies are not robust. In *Proceedings of the 29th ACM SIGACT Symposium on Theory of Computing* (El Paso, Tex., May).

Loui, M. C., and Abu-Amara, H. H. 1987. Memory requirements for agreement among unreliable asynchronous processes. *Adv. Comput. Res. 4*, 163–183.

Lynch, N., and Tuttle, M. 1988. An introduction to input/output automata. Tech. Rep. MIT/LCS/TM-373. Lab. Comput. Sci., Mass. Inst. Tech., Cambridge, Mass.

Moran, S., and Rappoport, L. 1996. On the robustness of h_m^r. In *Proceedings of the 10th Workshop on Distributed Algorithms* (Bologna, Italy, Oct.). Lecture Note in Computer Science, Vol. 1151. Springer-Verlag, New York, pp. 344–361.

PETERSON, G. L., BAZZI, R. A., AND NEIGER, G. 1994. A gap theorem for consensus types. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, Calif., Aug. 14–17). ACM, New York, pp. 344–353.

PLOTKIN, S. A. 1989. Sticky bits and universality of consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alb., Canada, Aug. 14–16). ACM, New York, pp. 159–175.

RACHMAN, O. 1994. Anomalies in the wait-free hierarchy. In *Proceedings of the 8th Workshop on Distributed Algorithms* (Terschelling, The Netherlands, Sept.-Oct.). Lecture Notes in Computer Science, vol. 857. Springer-Verlag, New York, pp. 156–163.

SAKS, M., AND ZAHAROGLOU, F. 1993. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing* (San Diego, Calif., May 16–18). ACM, New York, pp. 101–110.

SCHENK, E. 1997. The consensus hierarchy is not robust. (Brief Announcement). In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*. ACM, New York.

RECEIVED JANUARY 1995; REVISED APRIL 1997; ACCEPTED OCTOBER 1996