The Anonymous Consensus Hierarchy and Naming Problems

Eric Ruppert

Technical Report CSE-2006-11

December 29, 2006

Department of Computer Science and Engineering

4700 Keele Street North York, Ontario M3J 1P3 Canada

Abstract

This paper investigates whether the assumption of unique identifiers is essential for wait-free distributed computing using shared objects of various types. Algorithms where all processes are programmed identically and do not use unique identifiers are called anonymous. A variety of results are given about the anonymous solvability of two key problems, consensus and naming, in systems of various sizes. These problems are used to define measures of a type T's power to solve problems anonymously. These measures provide a significant amount of information about whether anonymous implementations of one type from another are possible. We compare these measures with one other and with the consensus numbers defined by Herlihy [14].

1 Introduction

It is routinely assumed, in the literature on distributed computing, that processes are equipped with unique identifiers or, equivalently, that each process can be given a different programme to follow. Such a system is called *eponymous* [23]. In contrast, in an *anonymous* system, processes do not have unique identifiers and are programmed identically. This paper studies the differences between anonymous and eponymous systems in the context of wait-free computation using shared memory. Thus, we take up Juliet's question: "What's in a name?" [28].

Unique identifiers are used for a variety of purposes in shared-memory systems. A process can announce or update information in a register that it alone is allowed to write, and that information will never be overwritten by another process. Process identifiers can be incorporated into timestamps to ensure that no two timestamps are identical, thereby avoiding the ABA problem. A compare&swap object (for example) can be accessed with two different arguments by two processes with different identifiers to allow the processes to determine which process accessed the object first. These techniques and many others that use unique identifiers are useful tools in solving problems. But are they truly essential? This paper studies how the answer to this question depends on the types of objects that are being used.

The primary motivation for this work is foundational: it is important to understand the significance of each assumption that is made when defining a model of distributed computing. In the widely-studied model of asynchronous, shared-memory computing where algorithms should be designed to tolerate failures, the ubiquitous assumption of unique identifiers has received scant attention.

Part of the reason that the assumption of unique names is so ingrained is the fact that they are available in most real systems (although this presupposes some architecture in which those names were handed out at some point, a task that may be quite difficult in systems where nodes frequently arrive and leave). It is worth knowing whether anonymous algorithms exist, even when identifiers are available, because the anonymous model of computation has its own advantages: simplicity and privacy. All other things being equal, an anonymous algorithm where every process runs its algorithm in the same way will be simpler to understand than an eponymous algorithm where all processes may execute entirely different sequences of steps, depending on their names. Privacy can be enhanced if processes running an anonymous algorithm access the shared memory carefully through a trusted proxy server that conceals their identities: the server containing the shared memory need not know who is accessing objects in the shared memory, or even whether two different accesses were performed by the same process.

This paper focusses on asynchronous shared-memory systems, where a collection of n processes communicate with one another using shared data structures (called objects) of various types. A type can be defined by a sequential specification, which describes how the object behaves when accessed by operations one at a time. In general, this specification may be nondeterministic: there may be several legal behaviours in response to a particular operation. The object's behaviour when accessed concurrently is constrained by the condition of linearizability [16]: each operation must appear to take place atomically some time during the interval of time between its invocation and its response. In keeping with the anonymous theme of this paper, we consider only *oblivious* object types, where the behaviour of the object in response to an operation cannot depend on the identity of the process that invoked the operation. The failure model allows any number of processes to experience crash failures; algorithms that work correctly in such an environment are called *wait-free* [14].

Herlihy identified the consensus problem as a key to understanding the ability of different types of shared objects to solve problems in the eponymous version of this model [14]. In the consensus problem, each process begins with an input value, and each non-faulty process produces an output value after a finite number of steps. The output values must be the same and the common output value must be the input value of some process. Herlihy defined the consensus number of a type T, denoted $\mathit{cons}(T)$, to be the maximum number of processes that can solve wait-free consensus using objects of type T and registers. If no such maximum exists, then $\mathit{cons}(T) = \infty$. This classifies objects into the *consensus hierarchy*: a type T is said to be at level k of the hierarchy if $\mathit{cons}(T) = k$. The consensus number of a type is an effective measure of its power in eponymous systems for two reasons. Firstly, consensus is *universal*: if $\mathit{cons}(T) \geq k$ then there is a wait-free eponymous implementation (for k processes) of *every* object from objects of type T and registers. Secondly, if $\mathit{cons}(T_1) < \mathit{cons}(T_2)$, then objects of type T_1 (and registers) cannot implement an object of type T_2 in an eponymous system of more than $\mathit{cons}(T_1)$ processes. Thus, the consensus hierarchy gives us a great deal of information about which implementations of one type from another are possible and which are not. In particular, this work influenced the design of shared-memory hardware to include objects from level ∞ of the consensus hierarchy.

A second key problem for studying the power of anonymous systems is the *naming problem*, where each process must output a distinct natural number (which it can then use as its name). One of the essential difficulties of solving some problems in anonymous systems is their inability to break symmetry: two processes with identical inputs may be scheduled in such a way that they take exactly the same sequence of steps. As we shall see, some types of shared objects can be used to break symmetry, while others cannot. The naming problem is essentially the problem of breaking symmetry between all pairs of processes.

The names chosen in a naming algorithm can be arbitrarily large. In the *strong naming problem*, each of the n processes must output a distinct number in the range $\{1, 2, \ldots, n\}$. We shall see that this version is strictly harder to solve. The naming and strong naming problems address the question of whether the identifiers that are used by so many algorithms can be assigned within the system model or whether they must be pre-assigned by the system designer. If the strong naming problem can be solved, then any eponymous algorithm can be run in an anonymous system: processes first simply choose identifiers and then run the code of that process. (Of course, this approach does not have the privacy benefits that some anonymous algorithms have; anonymity is necessary for true privacy, but not sufficient.)

The following assumptions are fairly standard when studying asynchronous shared-memory systems and are, in particular, used for Herlihy's definition of $\mathit{cons}(T)$:

(1) an unlimited number of objects of type T are available,
(2) an unlimited number of registers are available,
(3) algorithms are deterministic,
(4) objects can be initialized by the algorithm designer,
(5) n (or an upper bound on n) is known in advance,
(6) processes have unique identifiers, and
(7) the identifiers are $1, 2, \ldots, n$.

Variants of the hierarchy were defined by altering assumptions (1) and (2) [17], but it was ultimately agreed that these assumptions were indeed the most natural ones to use in defining the hierarchy. If assumption (3) is dropped, the consensus hierarchy collapses because there is a randomized algorithm to solve consensus among any number of processes using only registers

[2, 9]. Borowsky, Gafni and Afek showed that assumption (4) is essentially redundant, at least for deterministic objects, given the other assumptions [6]. However, the construction they used to prove this relies heavily on unique identifiers. Algorithms that work without assumption (5) are called *uniform* algorithms, and have been widely studied in the context of eponymous systems. (For examples, see the survey by Aguilera [1].) For this paper, we retain assumptions (1) to (5) in order to investigate the importance of assumptions (6) and (7). (In Section 4.1 we also briefly consider the effect of dropping assumption (5).) The significance of assumption (6) was questioned by Buhrman *et al.*, who showed that it was indeed crucial for Herlihy's universality result [8]. In particular, they studied systems that are equipped with registers and black-box objects that solve consensus. They showed that such a system cannot solve the naming problem. We continue this line of research by studying shared-memory systems equipped with arbitrary types of shared objects. Part of the goal of this paper is to understand what Herlihy's classification into the consensus hierarchy would have looked like without assumption (6). We shall also see whether assumption (7) is an essential addition to assumption (6).

1.1 Results

We define the *anonymous consensus number* of an object type T, denoted $acons(T)$, to be the maximum number of processes for which there exists an anonymous wait-free consensus algorithm using objects of type T and registers. If there is no such maximum, then $acons(T) = \infty$. Except for the requirement that the algorithm be anonymous, this definition is the same as Herlihy's definition of $cons(T)$. We also define the *anonymous consensus hierarchy* to be the classification of types according to their anonymous consensus numbers: a type is said to be at level k of this hierarchy if its anonymous consensus number is k.

We also define the *strong naming number* of an object type T, denoted $snaming(T)$ to be the maximum number of processes for which there exists an anonymous wait-free strong naming algorithm using objects of type T and registers. If there is no such maximum, then $snaming(T) = \infty$. There is no need to define a corresponding hierarchy for the ordinary naming problem, since we shall prove that any type that can be used to solve naming among two processes can solve naming among any number of processes. In the end, it may turn out that the strong naming hierarchy similarly collapses into two levels (1 and ∞): it is an open question whether there exist types at other levels of the strong naming hierarchy.

Herlihy showed that, in eponymous systems, a type T is universal for k processes if and only if $cons(T) \geq k$. The same result does not hold for anonymous systems. We define the universal number of a type T, denoted $univ(T)$, to be the maximum number of processes for which every type of object can be anonymously implemented from objects of type T and registers. If no such maximum exists, then $univ(T) = \infty$. We shall demonstrate that, if the system can solve *strong* naming for two processes, $univ(T) = cons(T)$. On the other hand, if the system cannot solve strong naming for two processes, then $univ(T)$ is clearly 1.

The classification of types T according to $acons(T)$, $univ(T)$ and their ability to solve the naming problem gives us a lot of information about whether anonymous implementations of one type from another are possible, just as the consensus hierarchy gives us information about eponymous implementations. By definition, an anonymous system with objects of type T and registers can implement any object for up to $univ(T)$ processes. If $acons(T_1) < acons(T_2)$ then T_1 (and registers) cannot implement T_2 anonymously in a system of more than $acons(T_1)$

Example type T	$cons(T)$	$acons(T)$	Solves naming?	$snaming(T)$	$univ(T)$
register	1	1	No	1	1
weak-name	1	1	Yes	1	1
?	1	1	Yes	$z \in \{2, 3, \ldots\}$	1
strong-name	1	1	Yes	∞	1
$T_{x,y}$	$x \geq 2$	$y \in \{2, 3, \ldots, x\}$	No	1	1
acons$_x$	$x \geq 2$	x	Yes	∞	x

Table 1: Classification of types according to ability to solve consensus and naming.

processes. Similarly, if T_2 can solve naming, but T_1 cannot, then T_1 cannot implement T_2 anonymously.

This paper proves the following facts about the anonymous consensus hierarchy.

- A type is at level 1 of the anonymous consensus hierarchy if and only if it is at level 1 of the (standard) consensus hierarchy (Theorem 3.5).
- For types T at level $x \geq 2$ of the consensus hierarchy, $acons(T)$ can take any value between 2 and x, inclusive (Proposition 3.4).
- If $acons(T) < cons(T)$ then objects of type T (and registers) cannot solve the naming problem (Theorem 5.3).

This paper proves the following results about the ability of types to solve naming.

- If naming can be solved for two processes, then it can be solved for any number of processes (Theorem 4.2). This theorem also provides a simple characterization of the types that can solve naming.
- If strong naming and consensus can be solved for two processes, then strong naming can be solved for any number of processes (Corollary 5.2).
- Strong naming is strictly harder than naming (Theorem 5.4).
- Strong naming is equivalent in difficulty to naming in systems where 2-process (eponymous) consensus can be solved (Theorem 5.1).
- Uniform naming is strictly harder than naming (Corollary 4.4).

Many of these results are summarized in Table 1: The results of this paper can be combined to show that every object type belongs to one of the rows in this table. (See Theorem 6.1.) It is not known whether any types fit into row 3. For each other row, we give examples of types belonging to the row. We also show that all deterministic types belong to rows 1, 5 and 6 of the table.

In Section 7 we also discuss the non-robustness of the classification of types according to $acons(T)$ and $univ(T)$.

1.2 Related Work

Asynchronous anonymous computation in *failure-free* models has been studied in several papers. Johnson and Schneider gave leader-election algorithms [18]. Attiya, Gorbach and Moran characterized tasks solvable using only registers [4]. Aspnes, Fich and Ruppert looked at models that provide other types of shared objects, such as counters [3]. They also saw how those models are related to an anonymous broadcast model.

The naming problem was introduced by Lipton and Park, who called it the processor identity problem [21]. If the system is failure-free, there are randomized algorithms that solve the problem using registers [10, 20, 21, 29].

There has been some recent work on fault-tolerant anonymous computing, which is closer to the topic of this paper. Panconesi *et al.* [26] gave a *randomized* wait-free naming algorithm, but it is not purely anonymous, since it requires using single-writer registers, which give the system some ability to distinguish between different processes' actions. Randomized naming is known to be impossible if only multi-writer registers are available [8, 10, 20]. However, consensus is solvable in this randomized model [8]. Thus, naming is strictly harder than consensus in this randomized setting.

Guerraoui and Ruppert investigated what can be implemented deterministically in an anonymous asynchronous system using only registers if processes may crash [13]. They gave a variety of algorithms for snapshots, timestamps and consensus, some of which are wait-free, while others satisfy the weaker non-blocking and obstruction-free progress properties. They also characterize the types that can be implemented in an obstruction-free manner from registers. Herlihy and Shavit [15] gave a characterization of decision tasks that have wait-free eponymous solutions using only registers, and extended the characterizaton to systems with a kind of anonymity: processes have identifiers but are only allowed to use them in very limited ways.

Merritt and Taubenfeld considered uniform algorithms (where processes do not know the size of the system) in a failure-free model where processes have identifiers but can only use them in a limited way: identifiers can be compared with one another but cannot be used, for example, to index into an array [24].

2 Preliminaries

The model of computation is a fairly standard one for shared-memory asynchronous distributed systems, except for the assumption of anonymity. We quickly describe the model here.

An object type is described by a *sequential specification* which is comprised of a set of possible states Q, a set of operations OP, a set of responses to operations RES and a transition function $\delta : Q \times OP \to \mathcal{P}(Q \times RES) - \emptyset$. (Here, $\mathcal{P}(S)$ denotes the power set of S.) If $(q', r) \in \delta(q, op)$, it means that if a process applies op to an object in state q, the object may return response r and switch to state q'. Since $\delta(q, op)$ may contain several different elements, objects are, in general, allowed to behave non-deterministically. An object type is called *deterministic* if $\delta(q, op)$ is a singleton set for all choices of q and op. An object type is said to have *finite non-determinism* if $\delta(q, op)$ is a finite set for all choices of q and op. The behaviour of an object when accessed concurrently is governed by the constraint that it is linearizable [16].

The most fundamental type of shared object is the (read-write) *register*. The state stores a value from some domain D and provides two deterministic operations: READ, which returns the value stored without changing it, and WRITE(v) (where $v \in D$), which changes the state of the register to v and returns ACK.

A distributed algorithm is a sequential programme for each process, $P_1, \ldots, P_n$. The subscripts used to identify the processes are used for convenience to reason about the distributed system; the processes themselves are unaware of them and the algorithm cannot make use of them. The programme can do standard (Turing-computable) steps on the process's local memory, and perform operations on shared objects. If the programmes assigned to all processes are identical, the algorithm is called *anonymous*. Otherwise, it is called *eponymous*.

A *step* of the algorithm is a single access to shared memory. It is denoted by a tuple (P_i, op, X, res, q), which specifies the process P_i that performs the operation, the operation op that is performed, the object X that it is performed on, the result res of the operation and the new state q that object X has immediately after the operation. There is no need to explicitly represent the local computations performed by process P_i: we may assume that P_i can carry out any necessary local computation immediately following each of its steps. An *execution* of an algorithm is a sequence of steps satisfying two constraints: the subsequence of steps performed on each object X must conform to that object's sequential specification, and the subsequence of steps performed by each process P_i must conform to P_i's programme. A *solo execution* by P_i is an execution where only P_i takes steps.

We consider two types of problems in this paper: one-shot tasks and long-lived implementations. In a *one-shot task*, each process receives an input (possibly null) and must eventually produce an output. The problem specification describes which outputs are legal for each possible assignment of inputs to processes. In a *long-lived implementation*, the goal is to implement or simulate an object X of type T. This requires giving a programme (to each process) for each operation that can be applied to X. Also, the initial state of all shared objects used in the implementation must be specified for each possible initial state of X. The implementation must be linearizable. In both cases, *wait-freedom* requires that there is no execution in which a process takes an infinite number of steps without completing its programme.

A *configuration* of an algorithm describes the state of all shared objects and the local state of all processes. If C is a configuration and s is a step that can be taken when the system is in configuration C, then Cs denotes the configuration that results from this step.

2.1 Consensus versus Binary Consensus

The consensus problem allows inputs to come from an arbitrary set. The *binary* consensus problem is a special case where the inputs are restricted to come from the set $\{0, 1\}$. The following proposition shows that the two problems are equivalent in anonymous systems.

Proposition 2.1 *If there is an anonymous binary consensus algorithm for k processes using objects of type T and registers, then $acons(T) \geq k$.*

The proposition is stated in a slightly different form as Proposition 8 of [13]: there it is given for obstruction-free algorithms, but the proof for wait-free algorithms is identical. The proof uses a fairly standard technique of agreeing on the output bit-by-bit.

Proposition 2.1 allows us to assume, throughout the rest of this paper, that all inputs to consensus are either 0 or 1. When referring to such input bits, we use the notation $\overline{x}$ to denote $1 - x$.

2.2 Valency

Some of the proofs in this paper use valency arguments, which originated in the work of Fischer, Lynch and Paterson [12] and have been used extensively since. (See [11] for a survey.) For some proofs that appear in this paper, the definitions of the terms that are usually used in valency arguments had to be generalized somewhat to take advantage of the anonymity of the system, so we introduce those generalized definitions here.

Fix some (binary) consensus algorithm. Suppose we have a tree T with the following properties. Each node represents a configuration of the algorithm. One configuration can be represented by several nodes. The root of the tree represents an initial configuration of the algorithm. If a node u is a child of another node v, then the configuration u must be reachable from v. An example of such a tree is the *complete* execution tree, in which each node v has one child for each configuration that is reachable from v by a single step.

A branch in T corresponds to an execution of the algorithm. If, during the execution that leads from the root to some node, a process outputs a value, label that node with the value. If no descendant of a node is labelled by $\overline{v}$, that node is called *v-valent in T*. A node that is 0-valent in T or 1-valent in T is called *univalent in T*. A node is called *multivalent in T* if it is not univalent in T.

When T is the complete execution tree, we omit the phrase "in T" for the terms defined in the previous paragraph. (For example, we say a node is multivalent rather than multivalent in T.) In this case, the definitions correspond to the original definitions of Fischer, Lynch and Paterson [12].

2.3 Wait-Freedom versus Bounded Wait-Freedom

In the definition of wait-freedom, every process is required to terminate after a finite number of its own steps. However, there is not necessarily a bound on how many steps it must execute. The following results show the existence of such a bound for one-shot tasks.

Lemma 2.2 (König's Lemma [19]) *If G is a connected graph with an infinite number of vertices and every vertex has finite degree, then there is an infinite path in G.*

The following application of König's Lemma was observed in [5, 7].

Corollary 2.3 *Consider a (deterministic) wait-free algorithm for a one-shot task. If the algorithm uses only objects with finite non-determinism then, for any input, the algorithm has a finite execution tree.*

Proof: Since the algorithm is wait-free, the execution tree has no infinite path. Since the objects have finite non-determinism, every node in the tree has a finite number of children. By König's Lemma, the tree is finite. ■

Throughout the remainder of this paper, we make the (realistic) assumption that all objects have finite non-determinism.

3 Anonymous Consensus Numbers

In this section, we investigate the anonymous consensus hierarchy. Since any anonymous consensus algorithm can also be used in an eponymous system with the same number of processes, we have the following observation.

Observation 3.1 *For all types T, $acons(T) \leq cons(T)$.*

One of the main applications of the anonymous consensus hierarchy is given by the following simple proposition.

Proposition 3.2 *If $acons(T_1) < acons(T_2)$, then T_2 cannot be implemented from objects of type T_1 and registers in an anonymous system of more than $acons(T_1)$ processes.*

Proof: Let $k = acons(T_1)$. Suppose there is an anonymous implementation of an object of type T_2 from objects of type T_1 and registers for $k+1$ processes. There is an anonymous consensus algorithm for $k+1$ processes using objects of type T_2 and registers. Each object of type T_2 in this consensus algorithm can be replaced by the implementation from objects of type T_1 and registers to yield an anonymous consensus algorithm for $k+1$ processes from registers and objects of type T_1. This contradicts the definition of k. ■

3.1 The *acons* Hierarchy is Full

The anonymous consensus hierarchy is *full*: it has types at every level. To see this, let $k > 1$ and consider the type acons$_k$, which supports the operation PROPOSE(x) for $x \in \{0,1\}$. Intuitively, if the object is suitably initialized, it returns the argument of the first PROPOSE operation to each of the first k PROPOSE operations that are performed on it. It returns to its initial state after the kth PROPOSE.

Formally, the object has state set $(\{0,1\} \times \{1,\ldots,k-1\}) \cup \{\perp\}$. Its (deterministic) transitions are described by the following function:

$$\begin{aligned}
\delta(\perp, \text{PROPOSE}(x)) &= \{((x,1),x)\}, \text{ for } x \in \{0,1\}, \\
\delta((x,i), \text{PROPOSE}(y)) &= \{((x,i+1),x)\}, \text{ for } x,y \in \{0,1\} \text{ and } 1 \leq i < k-1, \text{and} \\
\delta((x,k-1), \text{PROPOSE}(y)) &= \{(\perp,x)\}, \text{ for } x,y \in \{0,1\}.
\end{aligned}$$

Proposition 3.3 *For $k \geq 2$, $acons(acons_k) = cons(acons_k) = k$.*

Proof: To solve anonymous consensus among k processes, each process proposes its input value to an acons$_k$ object which is initially in state $\perp$ and outputs the value it returns. Thus, $acons(\text{acons}_k) \geq k$. A simple valency argument proves that $cons(\text{acons}_k) \leq k$. (This also follows from Proposition 12 in [27].) Suppose there is an eponymous consensus algorithm for $k+1$ processes that uses registers and acons$_k$ objects. By the usual valency argument [14, 22], there is a multivalent configuration C such that any step will take the system into a univalent configuration. Furthermore, each process accesses the same object X in the next step after C, and that object must be an acons$_k$ object. If X is in state $\perp$ in C, then let $i = 0$; otherwise, let i be the second component of the state of X in C. Let s_0 and s_1 be steps by some processes P_0 and P_1 after C such that Cs_0 is 0-valent and Cs_1 is 1-valent. Let C_0 be the 0-valent configuration reached from C by having processes $P_0, P_1, P_2, \ldots, P_{k-i-1}$ each take a step (in this order). Let C_1 be the 1-valent configuration reached from C by having processes $P_1, P_0, P_2, P_3, \ldots, P_{k-i-1}$ each take a step (in this order). Note that there is a process P_k that does not take a step in either of these sequences. Then X will be in state $\perp$ in both C_0 and C_1, because it has been accessed $k-i$ times after C. Each shared variable has the same state in C_0 and C_1. The internal state of P_k is the same in C_0 and C_1. So a solo execution by P_k from C_0 must output the same value as a solo execution by P_k from C_1, contradicting the fact that C_0 and C_1 have opposite valencies. Thus, $cons(\text{acons}_k) \leq k$. The proposition follows from Observation 3.1. ■

Levels 1 and ∞ of the anonymous consensus hierarchy contain the following two object types. A register has consensus number one [9, 22], so its anonymous consensus number is also one, by Observation 3.1. Herlihy gave a simple algorithm to show that a single compare&swap object can be used to solve consensus among any number of processes [14]. The object is initialized to $\perp$ and each process with input x executes $\textsc{Compare\&Swap}(\perp, x)$. The process returns x if the operation is successful, or the result of the operation otherwise. This algorithm is anonymous, so $acons(\text{compare\&swap}) = \infty$.

3.2 Comparing the *acons* and *cons* Hierarchies

Observation 3.1 says that anonymous consensus is no easier than consensus. Here, we show that anonymous consensus is strictly harder than eponymous consensus in some shared-memory models: it is possible for $acons(T)$ to be strictly smaller than $cons(T)$.

We define an object type $T_{x,y}$ that has $cons(T) = x$ and $acons(T) = y$, for any x and y satisfying $2 \leq y \leq x \leq \infty$. The type $T_{x,y}$ will, of necessity, be somewhat artificial, since we wish to construct an example for all possible values of x and y. However, as we shall see in Section 3.3, there are more natural objects that have $acons(T) < cons(T)$. Intuitively, an object of type $T_{x,y}$ simply solves the consensus problem by returning the first value given to it by a process, but only if it is accessed anonymously by at most y processes and by at most $y - x$ additional processes that use unique identifiers when accessing the object. If more than y processes try to access it without giving an identifier or if two processes use the same identifier, then the object becomes "upset" and returns useless random results to all further accesses. In order to implement this functionality, the state of the object remembers the value first proposed to it, the number of anonymous accesses that have taken place and the set of identifiers that have been used by the eponymous accesses.

Formally, the type $T_{x,y}$ has state set

$$\{\perp, \textsc{Upset}\} \cup \{(r, k, \mathcal{S}) : r \in \{0,1\}, 0 \leq k \leq y, \mathcal{S} \subseteq \{y+1, \ldots, x\} \text{ and either } k > 0 \text{ or } \mathcal{S} \neq \emptyset\}.$$

(If $y = \infty$, k will be a non-negative integer and $\mathcal{S}$ will always be $\emptyset$. If $y < x = \infty$, then $\mathcal{S}$ will be a subset of $\{y+1, y+2, \ldots\}$.) It provides two operations, $\textsc{Propose}(v)$ where $v \in \{0,1\}$ (an anonymous access) and (if $y < \infty$) $\textsc{Propose}(v, id)$ where $v \in \{0,1\}$ and $id \in \{y+1, \ldots, x\}$ (an eponymous access). The non-deterministic transition function is specified by

$$\begin{aligned}
\delta(\perp, \textsc{Propose}(v)) &= \{((v, 1, \emptyset), v)\}, \\
\delta(\perp, \textsc{Propose}(v, id)) &= \{((v, 0, \{id\}), v)\}, \\
\delta((r, k, \mathcal{S}), \textsc{Propose}(v)) &= \left\{ \begin{array}{ll} \{((r, k+1, \mathcal{S}), r)\} & \text{if } k < y, \\ \{(\textsc{Upset}, 0), (\textsc{Upset}, 1)\} & \text{if } k = y, \end{array} \right\}
\end{aligned}$$

$$\begin{aligned}
\delta((r,k,\mathcal{S}),\textsc{Propose}(v,id)) &= \left\{ \begin{array}{ll} \{((r,k,\mathcal{S}\cup\{id\}),r)\} & \text{if } id \notin \mathcal{S}, \\ \{(\textsc{Upset},0),(\textsc{Upset},1)\} & \text{if } id \in \mathcal{S}, \end{array} \right\} \\
\delta(\textsc{Upset},\textsc{Propose}(v)) &= \{(\textsc{Upset},0),(\textsc{Upset},1)\}, \text{ and} \\
\delta(\textsc{Upset},\textsc{Propose}(v,id)) &= \{(\textsc{Upset},0),(\textsc{Upset},1)\}.
\end{aligned}$$

Proposition 3.4 *For $2 \le y \le x \le \infty$, $cons(T_{x,y}) = x$ and $acons(T_{x,y}) = y$.*

Proof: If $x < \infty$, there is a simple eponymous consensus algorithm for x processes that uses a single object of type $T_{x,y}$, initialized to state $\perp$. For $1 \le i \le y$, the ith process performs a Propose(*input*) operation on the object and returns the result. For $y < i \le x$, the ith process performs a Propose(*input*, i) operation on the object. It is easy to see that the object will never become upset and every process will return the first input value that was proposed to the object. If $x = \infty$, then the preceding algorithm can be used with any number of processes. Thus, $cons(T_{x,y}) \ge x$.

If $y < \infty$, y processes can solve consensus anonymously as in the previous paragraph: each process performs a Propose(*input*) operation on an object initialized to $\perp$ and returns the result. If $y = \infty$, then this algorithm can be used for any number of processes. Thus, $acons(T_{x,y}) \ge y$.

If $x < \infty$, we can see that $cons(T_{x,y}) \le x$ using a straightforward valency argument. To derive a contradiction, suppose there is an eponymous consensus algorithm for $x+1$ processes using objects of type $T_{x,y}$ and registers. By the usual valency argument [14, 22], there is a multivalent configuration C such that any step from C leads to a univalent configuration. Furthermore, all processes must access the same object X of type $T_{x,y}$ in their next step after C. Let s_0 and s_1 be steps by different processes P and Q, respectively, such that Cs_0 is 0-valent and Cs_1 is 1-valent. Let α be the solo execution by P, starting from Cs_0. Since Cs_0 is 0-valent, P must output 0 in this execution. Let C' be the configuration that is obtained by starting from Cs_1 and having the $x-1$ processes other than P and Q each take one step, followed by one step, s_0, by P. This is legal because, regardless of the state of X in C, if $x+1$ processes access X, it will end up in the state Upset, and the object is free to return either 0 or 1 to the last of the $x+1$ accesses. Furthermore, α is a legal continuation from configuration C' since the local state of P and the state of every object except X is the same in Cs_0 and C', and it is possible for X to return the same sequence of responses to subsequent accesses by P because X is upset in C'. Thus, there is an execution from Cs_1 where P outputs 0, contradicting the fact that Cs_1 is 1-valent. This completes the proof that $cons(T_{x,y}) \le x$.

Finally, we show that $acons(T_{x,y}) \le y$. To derive a contradiction, suppose there is an anonymous consensus algorithm for $y+1$ processes, denoted $P_0, \ldots, P_y$, using only objects of type $T_{x,y}$ and registers. We adapt the valency technique to take advantage of the anonymity of the algorithm.

We construct a tree, where each node represents a configuration C of the algorithm such that process $P_1, \ldots, P_y$ have the same local state in C. The root is the initial configuration of the algorithm where process P_0 has input 0 and processes $P_1, \ldots, P_y$ have input 1. If C is any configuration in the tree where P_0 has not terminated, we add *left children* of C to represent the configurations that can be reached from C by a single step of process P_0. Similarly, if C is any configuration in the tree where P_1 has not terminated, we add *right children* of C to represent the configurations that can be reached from C if each of the processes $P_1, \ldots, P_y$ take an identical step and receive an identical response. There is at least one such extension because a sequence of WRITES to a register will all return a null response, a sequence of READS of a register will all return identical responses, and a sequence of operations on an object of type $T_{x,y}$ *can* always all return identical responses. Since the algorithm is wait-free, there are no infinite branches in the tree.

All leaves of T are univalent in T. The root is multivalent in T, since the executions where only P_0 takes steps must produce output 0 and the executions where P_0 takes no steps must produce output 1. There must be a node C such that C is multivalent in T and C's children are all univalent in T; otherwise there would be an infinite path of nodes that are multivalent in T. Then there must be a left child C_{left} of C that is v-valent in T and a right child C_{right} of C that is $\overline{v}$-valent in T.

If P_0 and P_1 either access different objects or if they both read the same register in their first steps after C then a right child of C_{left} and a left child of C_{right} are identical configurations, contradicting the fact that they have opposite valencies. If, in their next steps after C, P_0 writes a register R and P_1 accesses R, then P_0 cannot distinguish C_{left} from a left child of C_{right}, which is again a contradiction. A symmetric argument applies if P_0 reads a register R and P_1 writes to R in their next steps after C. Thus, P_0 and P_1 must both access the same object, X, in their next steps after C and X must be of type $T_{x,y}$.

Let α be a solo execution by P_0, starting from configuration C and passing through C_{left}. In this execution, P_0 must output v. If an object of type $T_{x,y}$ (in any state) has the same operation applied to it y times, it will end up in the state UPSET (since $y \geq 2$). Thus, in C_{right}, X must be in state UPSET, since it has just been accessed by y identical operations. The local state of P_0 and the state of every object except X are the same in C and C_{right}. Thus, the execution α is also legal starting from C_{right}, since the sequence of responses that P_0 receives from X in α can also occur if the execution is started from C_{right}, where X is upset. This means that C_{right} has a descendant in T that outputs v. This contradicts the fact that C_{right} is $\overline{v}$-valent in T. This contradiction proves that $acons(T_{x,y}) \leq y$, as required. ■

Although the consensus hierarchy and the anonymous consensus hierarchy are quite different, the division between levels one and two coincide.

Theorem 3.5 *For any type T, $cons(T) = 1$ if and only if $acons(T) = 1$.*

Proof: The "only if" direction follows from Observation 3.1.

To prove the other direction, it must be shown that $cons(T) \geq 2$ implies $acons(T) \geq 2$. Assume that $cons(T) \geq 2$. Let PROPOSE$_0(x)$ and PROPOSE$_1(x)$ be the code that is executed by two processes to solve consensus eponymously. Let B be a bound on the maximum number of steps a process must do while executing either of these routines. Corollary 2.3 guarantees the existence of such a bound, since there are only two possible inputs to each of the two processes in the binary consensus algorithm.

The algorithm in Figure 1 is an anonymous two-process consensus algorithm. It uses two registers R_0 and R_1, which are initially $\perp$, in addition to any shared objects used by PROPOSE$_1$ and PROPOSE$_2$. It is clearly wait-free. (It is necessary to include the "time limit" of B steps in calling the subroutine PROPOSE$_x(x)$ because there are some executions in which both processes will call PROPOSE$_0(0)$ or both will call PROPOSE$_1(1)$, and if this occurs there is no guarantee that those subroutines will halt.) If both processes have the same input x, then $R_{\overline{x}}$ always has the value $\perp$, so processes may only output x. For the remainder of the proof, suppose the two

```
PROPOSE($x$)
    if a READ of $R_{\overline{x}}$ returns $\top$ then
        return $\overline{x}$
    else
        WRITE($\top$) in $R_x$
        run PROPOSE$_x(x)$ until it halts or $B$ steps of it have been taken
        let $r$ be the result produced (if it halts)
        if a READ of $R_{\overline{x}}$ returns $\top$ then
            output $r$
        else
            output $x$
        end if
    end if
end PROPOSE
```

Figure 1: Anonymous consensus algorithm using eponymous consensus algorithm.

processes have different input values. It must be shown that, if both produce an output, those outputs are the same. Consider two cases.

If one process P with input x sees $\top$ when first reading $R_{\overline{x}}$, then the value of R_x remains $\perp$ throughout the execution. Thus, the other process, Q, can only output its own input value, $\overline{x}$. Process P also can only return $\overline{x}$.

If each of the processes receives $\perp$ as the result of its first READ, then one process runs PROPOSE$_0(0)$ and the other process runs PROPOSE$_1(1)$. This means that the subroutines will both terminate within B steps and will both produce the same output r. If both processes return r, agreement is guaranteed. However, if one process P sees $\perp$ in its second READ of $R_{\overline{x}}$, then it returns its own input x. In this case, P has completed running PROPOSE$_x(x)$ before the other process Q started running PROPOSE$_{\overline{x}}(\overline{x})$, so r must be equal to x, and Q must return x also. ■

3.3 Some Examples

Many of the object types discussed in this paper are somewhat artificially constructed in order to highlight the differences between the anonymous and eponymous model fully. In this section, we start with an example of a natural type where there is a stark difference between the anonymous and eponymous models.

Consider an array $A[1..m]$ whose elements can be read, written and atomically swapped. A SWAP(i, j) operation exchanges the values stored in $A[i]$ and $A[j]$ and returns an acknowledgement to the process that invoked the operation. Herlihy showed that this object has consensus number ∞ [14]. However, the type is much weaker in anonymous systems.

Proposition 3.6 *An array of registers with memory-to-memory* SWAP *has anonymous consensus number 2.*

Proof: It follows from Theorem 3.5 that the anonymous consensus number is at least 2. We now show that there is no three-process anonymous consensus algorithm using a valency argument. Suppose there was such an algorithm using arrays with memory-to-memory SWAP to derive a contradiction. We construct a tree T of executions whose root is the initial configuration where two processes, P and Q, have input 0 and process R has input 1. To each node in the tree representing a configuration C, we add a left child that represents the configuration that can be reached from C by having P and Q each take one step (if they have not terminated) and a right child that represents the configuration that can be reached from C by having R take one step (if it has not terminated). Note that, in every configuration in the tree, P and Q have the same local state, so they take identical steps in reaching the left child of that configuration. The root is multivalent in T, since the leftmost path must lead to the output 0 and the rightmost path must lead to the output 1. Thus, there is a critical configuration C that is multivalent in T and whose children are univalent in T. One child of C is 0-valent in T and the other is 1-valent in T.

As in the proof of Proposition 3.2, the next steps that are taken by P, Q and R after C must be on the same array, and they cannot be READS or WRITES. Thus, P and Q must perform a SWAP(i, j) and R must perform a SWAP(k, ℓ) on the array after C. Let C_1 be the right child of the left child of C and let C_2 be the right child of C. Then the configurations C_1 and C_2 are identical, contradicting the fact that they have opposite valencies in T. ■

Herlihy also considered an array $A[1..m]$ where entries can be read, written and copied. A COPY(i, j) command atomically changes the value stored in location $A[j]$ to the value that is stored in location $A[i]$. Herlihy showed that this type has consensus number ∞ [14]. However, his algorithm makes essential use of process identifiers. There is a much simpler anonymous algorithm for (binary) consensus that works for any number of processes.

Proposition 3.7 *An array of registers with memory-to-memory copy has anonymous consensus number* ∞.

Proof: There is an anonymous binary consensus algorithm that uses an array $A[0, 1]$ that is initialized with $A[0] = 0$ and $A[1] = 1$. Processes with input 0 execute a COPY$(0, 1)$ operation and then return the value they read from $A[1]$. Processes with input 1 execute a COPY$(1, 0)$ operation and then return the value they read from $A[0]$. In any execution, the first COPY operation ensures that both elements of the array contain the input value of the process that performed the COPY, and those values will never change thereafter, so processes can output only that value. ■

4 Naming

In this section, we take a closer look at the ability of different types of shared objects to solve the naming problem. Throughout Sections 4 and 5, all algorithms and systems are anonymous, since the naming and strong naming problems are trivial in the eponymous case. First, we see that the number of processes in the system has no effect on whether the naming problem is solvable. Theorem 4.2, below, shows that, in contrast to the consensus problem, solving naming among any number of processes is no harder than solving it among two processes. It also gives an exact characterization of the types of objects that can be used to solve the naming problem, using the following definition of Aspnes, Fich and Ruppert [3].

Definition 4.1 *An operation defined on a shared object type is called idemdicent if, for every starting state, for every operation, and for every choice of operands for that operation, it is possible that two consecutive invocations of the operation with these operands return identical responses. An object type is called idemdicent if every operation defined on it is idemdicent.*

Intuitively, an object is idemdicent if it is incapable of "breaking symmetry" between two processes that access it. Examples of idemdicent objects include registers, snapshot objects, resettable consensus objects, counters (with separate READ and INCREMENT operations, the latter of which returns a null result), and the type $T_{x,y}$ defined in Section 3.2.

Theorem 4.2 *For any type T and any $n \geq 2$, the following statements are equivalent.*

(1) *Naming can be solved for n processes using objects of type T.*

(2) *Naming can be solved for n processes using objects of type T and registers.*

(3) *Naming can be solved for two processes using objects of type T.*

(4) *Naming can be solved for two processes using objects of type T and registers.*

(5) *T is not idemdicent.*

Proof: First observe that the following implications are trivial: (1) $\Rightarrow$ (2) $\Rightarrow$ (4) and (1) $\Rightarrow$ (3) $\Rightarrow$ (4). Showing that (4) $\Rightarrow$ (5) and (5) $\Rightarrow$ (1) will complete the proof.

(4) $\Rightarrow$ (5): Suppose that there is a naming algorithm for two processes that uses objects of type T and registers. To derive a contradiction, assume T is idemdicent. Consider an execution of the naming algorithm where the two processes alternate taking steps, and they both perform the same sequence of operations and get the same sequence of responses. This is possible, since the algorithm is anonymous and every time the two processes perform the same operation on an object, that object can return the same response to both of them, by the definition of idemdicence. In this execution, both processes must produce the same output, contradicting the assumption that the algorithm solves the naming problem.

(5) $\Rightarrow$ (1): Let T be a type that is not idemdicent. Then there is some state q and some operation op such that the first two invocations of op on an object in state q cannot return the same response. Let $\mathcal{R}$ be the set of possible responses that can be returned if op is performed up to n times on an object initialized to state q. Since T has finite non-determinism, $\mathcal{R}$ is a finite set. Let $d = |\mathcal{R}|$.

An algorithm for the naming problem among n processes can be constructed using objects of type T as a weak kind of splitter [25]. The naming algorithm uses a tree of height $n-1$, where every internal node has d children. Each node has an associated object of type T, initialized to state q. The edges leading from a node v to its d children are labelled by the elements of $\mathcal{R}$. The leaves of the tree are labelled by distinct natural numbers.

To run the naming algorithm, each process starts at the root of the tree and follows a path towards a leaf. For each internal node v that the process visits, it applies the operation op to the associated object. When it receives a response r, the process advances to a child of v along the edge labelled by r. When the process reaches a leaf, it outputs the label of that leaf.

If m processes accesses a node v, the first two processes must receive different responses and proceed to different children of v. Thus, at most $m-1$ processes will visit any child of v. It follows by induction on k that at most $n-k$ processes will visit any node at depth k in the tree. Thus, at most one process will reach any leaf, and this guarantees that the names produced by the algorithm are distinct. ■

4.1 Uniform Naming

The naming algorithm used to prove that (5) implies (1) in the proof of Theorem 4.2 is not uniform: a process must know n (or at least an upper bound on n) because the process must proceed through the tree until it reaches a node at depth $n-1$. This knowledge is indeed necessary: we show in this section that some types that can solve naming for any number of processes cannot solve uniform naming.

Consider the toggle object, which stores a single bit and provides a single operation, called TOGGLE, that changes the value of the bit and returns its new value. This object is not idemdicent, so it is capable of solving the naming problem among any number of processes, by Theorem 4.2. However, we shall show that there is no uniform naming algorithm. To do this, we first prove the following lower bound on the time complexity of any naming algorithm that uses toggle objects and registers.

Proposition 4.3 *In any naming algorithm for n processes that uses toggle objects and registers, the number of* TOGGLE *operations performed during any solo execution of the algorithm is greater than* $\log_2(n-1)$.

Proof: Consider any naming algorithm for n processes that uses toggle objects and registers. Let α be any solo execution of the algorithm by a single process p. The execution α can be broken up into segments $\alpha = \alpha_0 t_1 \alpha_1 t_2 \alpha_2 \cdots t_k \alpha_k$, where each t_i is a single TOGGLE step on a toggle object X_i and each α_i is a (possibly empty) sequence of READS and WRITES.

To derive a contradiction, assume $k \le \log_2(n-1)$. Then $n \ge 2^k + 1$. We construct an execution where $2^k + 1$ processes begin running in lockstep, each performing a prefix of the sequence of steps that p does in α. Each time p performs a TOGGLE, we throw away the half of the processes who receive the opposite answer to p, so that all remaining processes can still continue taking steps as clones of p. Consider the execution $\beta_0 \tau_1 \beta_1 \tau_2 \beta_2 \cdots \tau_k \beta_k$, where

- β_0 is an execution by $2^k + 1$ processes running in lock-step, so that each process performs the same sequence of steps as p performs in α_0,
- τ_i is an execution where each of the $2^{k-i+1} + 1$ processes that took steps in β_{k-1} perform a single operation, which will be a TOGGLE on X_i, and
- β_i is an execution where each of the $2^{k-i} + 1$ processes that received the same response from X_i during τ_i as p received in step t_i run in lockstep; each process performs the same sequence of steps as p performs in α_i.

We remark that this execution is legal, since the state of each shared object is the same at the end of the executions $\alpha_0 t_1 \alpha_1 \cdots t_i \alpha_i$ and $\beta_0 \tau_1 \beta_1 \cdots \tau_i \beta_i$ for all $i \ge 0$. The two processes that take steps in α_k have taken exactly the same sequence of steps as p does in α, so they must both terminate and output the same name as p chose in α. This violates the name uniqueness property of the naming problem. Thus, the assumption that $k \le \log_2(n-1)$ must be false. ■

Corollary 4.4 *There is no uniform naming algorithm using toggle objects and registers.*

Proof: Suppose there is such an algorithm. Consider a solo execution of the algorithm. Let m be the number of steps the process takes in this execution. This algorithm solves the naming problem in a system of $2^{m+1} + 1$ processes. According to the preceding proposition, a solo execution in such a system must take at least $m + 1$ steps, contradicting the definition of m. ■

Thus, the uniform naming problem is strictly harder than the naming problem.

5 Strong Naming

We now consider the strong naming problem, where processes must return distinct names from the range $\{1, \ldots, n\}$. Below, in Corollary 5.2, we obtain a result analogous to Theorem 4.2 for the strong naming problem. However, it applies only to object types whose consensus numbers are at least two. It is an open problem whether the result also holds for objects at level one of the consensus hierarchy. The following proposition shows that, if the system is capable of solving consensus among two processes, the naming and strong naming problems are equivalent.

Theorem 5.1 *If $cons(T) \geq 2$ then, for any n, objects of type T and registers can be used to solve naming for n processes if and only if they can be used to solve strong naming for n processes.*

Proof: The claim is trivial for $n = 1$, so assume $n \geq 2$. The "if" part of the claim is trivial since any strong naming algorithm is also a naming algorithm.

To prove the converse, suppose that naming is solvable for n processes using objects of type T and registers. By Theorem 4.2, T is not idemdicent. By Corollary 2.3, the tree of possible executions of the naming algorithm is finite, and therefore the set of possible names is finite. Let M be the maximum possible name.

The following algorithm solves the strong naming problem for n processes. It uses a data structure that consists of n binary trees, each with M leaves, to implement a renaming algorithm that reduces the size of the name space. The trees are numbered 1 to n. Each internal node of each tree is associated with a different instance of two-process eponymous consensus, which is implemented from objects of type T and registers. Each process first obtains a name $i \in \{1, 2, \ldots, M\}$ using the naming algorithm. It then accesses the first binary tree, starting from the ith leaf and moving along the path from that leaf to the root. At each internal node, it proposes *left* or *right* to the instance of the two-process eponymous consensus algorithm associated with the node. If the process arrived at the node from its left child, it proposes *left*, using the consensus algorithm for process 1, and if it arrived from the right child, it proposes *right*, using the consensus algorithm for process 2. The process continues towards the root only if the result returned is equal to the value it proposed. In this case, we say that the process *wins* at that node. A process that does not receive its own input as the output of consensus at some node is said to *lose* at that node. If it ever loses at some node, it stops accessing the tree and switches to the next tree. It accesses this tree in exactly the same way, again moving on to the next one if it ever loses at some node. The process continues accessing trees in this way until it wins at the root of one of the trees. If the process wins at the root of the jth tree, it outputs j as its name and halts.

The consensus algorithm associated with a node can be run by at most one process for each of the two children of the node, namely the process that either started at that child (if the child is a leaf) or the process that won at that child (if the child is an internal node). Thus, the algorithm executed at each node will correctly solve consensus. At most one process will win at the root of any tree, and it follows that all names produced will be distinct elements of the set $\{1, 2, \ldots, n\}$. To prove that processes terminate, we observe that if r processes access nodes at some level of tree T, at least $\lceil r/2 \rceil$ of them either win or experience a halting failure. So, if any processes access tree T, at least one process either wins at the root of T or fails at some time during its accesses to T. Thus, if k processes access a tree, at most $k-1$ processes access the next tree. It follows that no process continues past the nth tree. Thus, every non-faulty process eventually outputs a name. ∎

Corollary 5.2 *For any type T with $cons(T) \geq 2$ and any $n \geq 2$, the following are equivalent.*

(1) *Strong naming can be solved for n processes using objects of type T and registers.*

(2) *Strong naming can be solved for 2 processes using objects of type T and registers.*

(3) *T is not idemdicent.*

Proof: This follows directly from Theorem 4.2 and Theorem 5.1. ∎

This corollary can be used to reveal another connection between the consensus problem and naming.

Theorem 5.3 *If T is not idemdicent, $acons(T) = cons(T)$.*

Proof: To derive a contradiction, suppose T is not idemdicent but $acons(T) \neq cons(T)$. By Observation 3.1, $acons(T) < cons(T)$. Thus, $cons(T) \geq 2$ and $acons(T) \neq \infty$. Since T is not idemdicent, strong naming can be solved for $acons(T) + 1$ processes using objects of type T and registers, by Corollary 5.2. Then, $acons(T) + 1$ processes can solve consensus anonymously using objects of type T and registers by first running the strong naming algorithm and then running the eponymous consensus algorithm for $acons(T) + 1$ processes. This contradicts the definition of $acons(T)$. ∎

Theorem 5.1 showed that the naming and strong naming are equivalent if the underlying system can solve two-process consensus. However, if this is not the case, strong naming is strictly harder than naming. We now define an object that can solve naming for any number of processes but cannot solve strong naming even for two processes.

The *weak-name* object has a single operation, GetName, that behaves as follows. The first two GetName operations non-deterministically return any two distinct names from the set $\{1, 2, 3\}$. If any further GetName operations are performed, the object non-deterministically chooses any value from the set $\{1, 2, 3\}$ to return.

More formally, the state set is $\{\perp, 1, 2, 3, \textsc{Upset}\}$. The transition function is given by

$$\begin{aligned}
\delta(\perp, \textsc{GetName}) &= \{(i, i) : i \in \{1, 2, 3\}\} \\
\delta(i, \textsc{GetName}) &= \{(\textsc{Upset}, j) : j \neq i, j \in \{1, 2, 3\}\} \\
\delta(\textsc{Upset}, \textsc{GetName}) &= \{(\textsc{Upset}, j) : j \in \{1, 2, 3\}\}
\end{aligned}$$

Theorem 5.4 *Weak-name objects can solve naming for n processes, for all n. Weak-name objects and registers cannot solve strong naming for two processes.*

Proof: The weak-name type is not idemdicent: if it is initialized to the state $\perp$, the first two accesses to it must return different results. By Theorem 4.2, it can solve naming for any number of processes.

Assume that weak-name objects and registers can solve strong naming for two processes. We shall use a reduction involving the renaming problem to derive a contradiction from this assumption. The *renaming* problem is to design an anonymous algorithm such that, in any execution where processes receive distinct inputs in the range $\{1, \ldots, M\}$, they output distinct values in the range $\{1, \ldots, m\}$. We describe how to build a two-process renaming algorithm for $M = 3$ and $m = 2$ using only registers. This was shown to be impossible by Herlihy and Shavit [15].

To solve the renaming problem, each of the two processes runs the strong naming algorithm. Each weak-name object X used in the naming algorithm is simulated without doing any accesses to shared memory as follows. We consider several cases, depending on how X is initialized. First, suppose X is initially $\perp$. When a process with input i is supposed to first access X, it pretends that the response from X was i. If the process does any subsequent accesses to X, it pretends X's response was $i \bmod 3 + 1$. Note that, in any execution, the first two simulated responses from X will be distinct, whether those two accesses are by the same process or by different processes. If X initially has state i, then all accesses to X return the response $i \bmod 3 + 1$. If X is initially UPSET, then all accesses to X return the response 1. This simulation of the algorithm requires only registers. Because this is a faithful simulation of the strong naming algorithm, the two processes will output distinct values from $\{1, 2\}$, thereby solving the renaming problem, which is impossible. ■

Corollary 5.5 *The type weak-name has consensus number 1.*

Proof: By Theorem 5.4, weak-name objects can solve the naming problem for two processes. If *cons*(weak-name) were bigger than one, then, by Theorem 5.1, strong naming for two processes could be solved using weak-name objects and registers, but this would contradict Theorem 5.4. ■

6 Classifying Object Types

The preceding results provide enough information to give a fairly complete picture of the classification of object types according to their ability to solve consensus and naming, and their universal numbers. This classification is given in the following theorem. Examples of types in the various categories will be described below.

Theorem 6.1 *Every type T belongs to one of the rows in Table 1 on page 5. (The fourth column describes whether naming can be solved for any number of processes greater than 1 using objects of type T and registers.)*

Proof: Consider any type T. We show that T fits into one of the rows in Table 1, ignoring the last column for the moment.

First, suppose *cons*$(T) = 1$. Then, by Observation 3.1, *acons*$(T) = 1$. If T is idemdicent, then it cannot solve naming (even with registers), by Theorem 4.2, and therefore its strong naming number is 1. Thus, T belongs to row 1 of the table. If T is not idemdicent, then it can solve naming, so it belongs to row 2, 3 or 4, depending on its strong naming number.

Now suppose *cons*$(T) = x > 1$. If T is idemdicent, then objects of type T (and registers) cannot solve naming, by Theorem 4.2. Therefore, they cannot solve strong naming, even for two processes, either. The anonymous consensus number of T must be at least two, by Theorem 3.5, and at most x, by Observation 3.1. Thus T belongs to row 5 of Table 1. If T is not idemdicent, then *acons*(T) is also x, by Proposition 5.3. Objects of type T can solve naming, by Theorem 4.2. Objects of type T and registers can solve strong naming for any number of processes, by Proposition 5.2. Thus, T belongs to row 5 of Table 1.

Finally, we show that the value given for *univ*(T) is correct for each row. For the first 4 rows, *univ*(T) must be 1, since objects of type T and registers cannot implement consensus for two processes. For row 5, *univ*(T) must be 1, since objects of type T and registers cannot solve strong naming for two processes. For types in row 6, *univ*$(T) \leq x$, since objects of type T and registers cannot implement consensus for $x + 1$ processes. To see that *univ*$(T) = x$, we describe how x processes can anonymously implement any object using objects of type T and registers. First, the processes solve strong naming and then, because *cons*$(T) = x$, they can apply Herlihy's eponymous universal construction to implement any object [14]. ■

It is an open question whether there exist any types belonging to row 3 of Table 1, so it is possible that this row could be removed from the table, or additional constraints on the value of z could be included. We now show that the rest of the classification cannot be improved: we give examples for each other row in Table 1.

A register has consensus number 1 [9, 22] and is idemdicent, so it belongs to row 1 of Table 1. The weak-name object, defined in Section 5, belongs to row 2 of Table 1, according to Theorem 5.4 and Corollary 5.5.

For row 4, we define a new *strong-name* type. It provides a single operation, GetName(k) where k is a positive integer, that returns a positive integer. If processes perform up to k GetName operations with the same argument k, the object returns distinct responses from $\{1, 2, \ldots, k\}$. If processes access the object in a different way, the object becomes upset and returns non-deterministic results.

More formally, the state set of the strong-name object is $\{\perp, \textsc{Upset}\} \cup \{(k, \mathcal{S}) : k \geq 1, \mathcal{S} \subseteq \{1, \ldots, k\}\}$. The state $(k, \mathcal{S})$ is intended to represent the situation where processes have been accessing the object with argument k and the object has already given the responses in the set $\mathcal{S}$. The transition function is defined by

$$\begin{aligned}
\delta(\perp, \textsc{GetName}(k)) &= \{((k, \{i\}), i) : 1 \leq i \leq k\}, \\
\delta((k, \mathcal{S}), \textsc{GetName}(k)) &= \{((k, \mathcal{S} \cup \{i\}), i) : i \in \{1, \ldots, k\} - \mathcal{S}\} \text{ if } |\mathcal{S}| < k, \\
\delta((k, \mathcal{S}), \textsc{GetName}(j)) &= \{(\textsc{Upset}, i) : 1 \leq i \leq j\} \text{ if } |\mathcal{S}| = k \text{ or } j \neq k, \text{ and} \\
\delta(\textsc{Upset}, \textsc{GetName}(k)) &= \{(\textsc{Upset}, i) : 1 \leq i \leq k\}.
\end{aligned}$$

The following proposition shows that this type occupies line 4 of Table 1.

Proposition 6.2 *A single strong-name object can solve strong naming for any number of processes and cons(strong-name)* $= 1$.

Proof: There is a very simple algorithm for strong naming among n processes using a single strong-name object initialized to the state $\perp$: each process performs a GETNAME(n) and outputs the object's response.

We now show that $cons$(strong-name) = 1. We do this by showing that a strong-name object can be implemented (eponymously) for two processes without using shared memory. So, if there were a two-process eponymous consensus algorithm that uses strong-name objects and registers, then two processes could solve eponymous consensus using only registers, which is impossible [14, 22].

A strong-name object X can be implemented in a two-process eponymous system without using shared memory at all as follows. If X is initialized to UPSET, all operations on X simply return 1. If X is initialized to $(k, \mathcal{S})$ let $\mathcal{S}' = \{1, \ldots, k\} - \mathcal{S}$. Any GETNAME($j$) with $j \neq k$ can simply return 1. Process 1's GETNAME(k) operations performed on X return the elements of $\mathcal{S}'$, in order. Process 2's GETNAME(k) operations performed on X return the elements of $\mathcal{S}'$ in reverse order (starting with the largest element). If either process runs out of values, it can start simply returning 1. Then, the only way that two GETNAME(k) operations on X can return the same result is if at least $k - |\mathcal{S}| + 1$ accesses to X have been done, by which time X should be in the UPSET state, and it is legal for equal results to be returned. If X is initialized to $\perp$, and a process's first operation on X is GETNAME(k), then the process can implement X as above, using $\mathcal{S}' = \{1, \ldots, k\}$. ∎

The object $T_{x,y}$, defined in Section 3.2, was shown in Proposition 3.4 to have $cons(T_{x,y}) = x$ and $acons(T_{x,y}) = y$ for every x and y satisfying $2 \leq y \leq x \leq \infty$. It is easy to verify that $T_{x,y}$ is idemdicent by examination of the transition function. If the object is in state $\perp$, the next two operations can return the first argument of the first operation. If the object is in state (r, k, S), the next two operations can both return r. If the object is in state UPSET, the next two operations can both return 0. Thus, objects of type $T_{x,y}$ cannot be used with registers to solve the naming problem. This means that there are types in row 5 of Table 1 for all possible values of x and y. The memory-to-memory swap array discussed in Section 3.3 is another example of a type in row 5 of the table.

The final row of Table 1 is occupied by the object acons$_x$, defined in Section 3.1, for $2 \leq x < \infty$. Notice that this object is not idemdicent, and can therefore solve naming, because two successive invocations of PROPOSE(1) starting from state $(0, x-1)$ must return 0 and 1. The compare&swap object also occupies the final row of Table 1, with $x = \infty$.

6.1 Deterministic Types

If we restrict attention to deterministic types only, then the classification of Theorem 6.1 can be refined. The objects given above as examples for rows 1 and 6 of Table 1 are deterministic. The following proposition rules out the possibility of any deterministic types in rows 2, 3 and 4 of Table 1.

Proposition 6.3 *If T is deterministic and not idemdicent, then $acons(T) \geq 2$.*

Proof: Since T is deterministic and not idemdicent, there is a state q and an operation op such that two successive invocations of op return results r_1 and r_2, respectively, with $r_1 \neq r_2$. Then the algorithm in Figure 2 solves consensus for two processes anonymously using one object

```
Propose($x$)
    Write($\top$) in $R_x$
    if applying $op$ to $X$ returns $r_2$ and a Read of $R_{\overline{x}}$ returns $\top$ then
        return $\overline{x}$
    else
        return $x$
    end if
end Propose
```

Figure 2: Two-process anonymous consensus using a deterministic non-idemdicent object.

X of type T, initialized to state q, and two registers R_0 and R_1, initialized to state $\bot$. In the algorithm, both processes can only return the input value of the first process that accesses X. ■

There are some deterministic types that belong to row 5 of Table 1. For example, standard x-consensus objects have consensus number x, but are idemdicent. It is an open question whether there is a type in row 5 for all possible values of x and y.

7 Non-robustness

Jayanti raised the question of the *robustness* of the consensus hierarchy [17]. In general, a robustness property means that two types of objects that cannot be used individually to solve a problem cannot be used together to do so. In particular, Jayanti asked whether two types at level k or lower in the consensus hierarchy can be used together to solve consensus among more than k processes. (It is known that the answer is yes, although this cannot happen for certain classes of objects. See [11] for a survey of these results.) We can ask the same question for the anonymous consensus hierarchy.

Theorem 7.1 *The anonymous consensus hierarchy is not robust: for $2 \leq y < x$, neither the weak-name nor the $T_{x,y}$ type can be used (with registers) to solve consensus anonymously among more than y processes, but anonymous consensus can be solved among x processes using objects of type weak-name, $T_{x,y}$ and registers together.*

Proof: It was shown in Corollary 5.5 that $cons$(weak-name) = 1, so $acons$(weak-name) = 1 by Observation 3.1. By Proposition 3.4, $acons(T_{x,y}) = y$.

Since the weak-name object is not idemdicent, x processes can choose unique names using them, by Theorem 4.2. Once the processes have chosen unique names, they can use objects of type $T_{x,y}$ and registers to implement the renaming algorithm used in the proof of Proposition 5.1. This will provide each process with a unique name in the range $\{1, \ldots, x\}$. Then the processes can use these names to run the eponymous consensus algorithm built from objects of type $T_{x,y}$ and registers. (Proposition 3.4 proved that such an algorithm exists.) ■

It follows from the argument above that the classification of types T according to $univ(T)$ is not robust either.

Proposition 7.2 *The types weak-name and $T_{x,y}$ each have universal number 1, but together the types weak-name, $T_{x,y}$ and registers can be used anonymously to implement any object type for x processes.*

Proof: It follows from Corollary 5.5 that *univ*(weak-name) = 1. As shown in Section 6, type $T_{x,y}$ is idemdicent, so it has $univ(T_{x,y}) = 1$.

The preceding proof showed that the types could be used together to solve strong naming and consensus among x processes. Then Herlihy's universal construction [14] can be used to implement any other object type in a system of x processes. ■

The same object types can be used to show that the strong naming hierarchy is not robust: we have $snaming(T_{2,2}) = snaming(\text{weak-name}) = 1$, but a system with both types (and registers) can solve strong naming for any number of processes, as shown in Theorem 5.1.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada and was partly done during a sabbatical at the École Polytechnique Fédérale de Lausanne, Switzerland.

References

[1] Marcos K. Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News*, 35(2), pages 36–59, June 2004.

[2] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2–3), pages 165–175, September 2003.

[3] James Aspnes, Faith Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3), pages 209–219, February 2006.

[4] Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2), pages 162–183, March 2002.

[5] Rida A. Bazzi, Gil Neiger, and Gary L. Peterson. On the use of registers in achieving wait-free consensus. In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 354–362, 1994.

[6] Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 363–372, 1994.

[7] Hagit Brit and Shlomo Moran. Wait-freedom vs. bounded wait-freedom in public data structures. *Journal of Universal Computer Science*, 2(1), pages 2–19, 1996.

[8] Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitanyi. On the importance of having an identity or, is consensus really universal? *Distributed Computing*, 18(3), pages 167–176, February 2006.

[9] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[10] Ömer Eğecioğlu and Ambuj K. Singh. Naming symmetric processes using shared variables. *Distributed Computing*, 8(1), pages 19–38, 1994.

[11] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3), pages 121–163, September 2003.

[12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), pages 374–382, April 1985.

[13] Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*. To appear.

[14] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), pages 124–149, January 1991.

[15] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6), pages 858–923, November 1999.

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

[17] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pages 145–157, 1993.

[18] Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proc. 4th ACM Symposium on Principles of Distributed Computing*, pages 13–22, 1985.

[19] Dénes König. Über eine Schlussweise aus dem Endlichen ins Unendliche. *Acta Litterarum ac Scientiarum Regiae Universitatis Hungaricae Francisco-Josephinae: Sectio Scientiarum Mathematicarum*, 3, pages 121–130, 1927. The required result is also stated as Theorem 3 in chapter VI of Dénes König, *Theory of Finite and Infinite Graphs*, Birkhäuser, Boston, 1990.

[20] Shay Kutten, Rafail Ostrovsky, and Boaz Patt-Shamir. The Las-Vegas processor identity problem (How and when to be unique). *Journal of Algorithms*, 37(2), pages 468–494, November 2000.

[21] Richard J. Lipton and Arvin Park. The processor identity problem. *Information Processing Letters*, 36(2), pages 91–94, October 1990.

[22] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, ed., *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, Connecticut, 1987.

[23] Marios Mavronicolas, Loizos Michael, and Paul Spirakis. Computing on a partially eponymous ring. In *Proc. 10th International Conference on Principles of Distributed Systems*, 2006. To appear.

[24] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes under assumptions on concurrency and participation. In *Distributed Computing, 14th International Conference*, pages 164–178, 2000.

[25] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1), pages 1–39, October 1995.

[26] Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3), pages 113–124, August 1998.

[27] Eric Ruppert. Determining consensus numbers. *SIAM Journal on Computing*, 30(4), pages 1156–1168, 2000.

[28] William Shakespeare. The Most Excellent and Lamentable Tragedy of Romeo and Juliet, II, ii, 1599.

[29] Shang-Hua Teng. Space efficient processor identity protocol. *Information Processing Letters*, 34(3), pages 147–154, April 1990.