From Bounded to Unbounded Concurrency Objects and Back

Yehuda Afek
afek@post.tau.ac.il

Adam Morrison
adamx@post.tau.ac.il

Guy Wertheim
vgvertex@gmail.com

School of Computer Science
Tel Aviv University

ABSTRACT

We consider the power of objects in the unbounded concurrency shared memory model, where there is an infinite set of processes and the number of processes active concurrently may increase without bound. By studying this model we obtain new results and observations that are relevant and meaningful to the standard bounded concurrency model.

First we resolve an open problem from 2006 and provide, contrary to what was conjectured, an unbounded concurrency wait-free implementation of a `swap` object from 2-consensus objects. This construction resolves another puzzle that has eluded us for a long time, that of considerably simplifying a 16 year old complicated bounded concurrency `swap` construction.

A further insight to the traditional bounded concurrency model that we obtain by studying the unbounded concurrency model, is a refinement of the top level of the wait-free hierarchy, the class of infinite-consensus number objects. First we resolve an open question of Merritt and Taubenfeld from 2003, showing that having n-consensus objects for all n does *not* imply consensus under unbounded concurrency. I.e., consensus alone, treated as a black box, *cannot* be "boosted" in this way. We continue to show an infinite-number consensus object that while able to perform consensus for any n-bounded concurrency (n unknown in advance) cannot solve consensus in the face of unbounded concurrency. This divides the infinite-consensus class of objects into two, those that can solve consensus for unbounded concurrency, and those that cannot.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*

General Terms

Algorithms, Theory

Keywords

Common2, Consensus, Swap, Unbounded concurrency, Wait-free

1. INTRODUCTION

We consider the power of objects in the wait-free unbounded concurrency model [10,18] that consists of a shared memory system with an infinite set of processes in which the number of concurrently active processes may increase without a bound. In addition to broadening our understanding of the limits of distributed computation, our unbounded concurrency results lead to new insights on the traditional bounded concurrency shared memory model. Our motivation is to improve the understanding of shared objects' wait-free computational power beyond what is possible with the classic characterization based on an object's *consensus number* [13,15], since it lumps together all objects with the *same* consensus number, providing little insight on the relationship between such objects. Here we use unbounded concurrency to provide some distinction between the objects with infinite consensus number.

In [2], Common2, the class of objects wait-free constructible from 2-consensus objects, was extended to the unbounded concurrency model. It was shown there that all previously known Common2 objects are also in unbounded concurrency Common2 with one exception: the `swap` object, which was conjectured in [2] to be impossible to implement in unbounded concurrency — hinting that `swap` is inherently more difficult to implement than, say, `fetch-and-add`. Here we show this is not the case, presenting an unbounded concurrency wait-free implementation of `swap` from 2-consensus objects, thus refuting the conjecture of [2]. This indicates that bounded concurrency Common2 and unbounded Common2 are possibly the same. Interestingly, despite the harder requirements, our unbounded concurrency `swap` implementation is *considerably simpler* than the previous (16-year old) n-process (bounded concurrency) algorithm [3, 21]. This demonstrates an appealing property of the unbounded concurrency model, which is part of the motivation for studying this model in the first place: it provides constructions that deal with the essence of the problem, and not with the artifacts of n, the *a priori* bound on the number of active processes. This yields simpler and cleaner algorithms and proofs.

A further insight we obtain on the traditional model concerns the nature of infinite-consensus number objects. We embark by resolving an open problem of Merritt and Taubenfeld from [19], showing that having n-consensus objects for all n does not imply consensus under unbounded concurrency. Next, we consider a natural extension of the Common2 question to the top level of the wait-free hierarchy: Does an infinite-consensus number object, which can solve consensus for any n, necessarily also solve consensus in the face of unbounded concurrency (i.e., $n = \infty$)? We show that there is an infinite-consensus object that while able to perform consensus for any n-bounded concurrency (even if n is unknown in advance) cannot solve consensus in the face of unbounded concurrency. This divides the infinite-consensus class of objects into two, those that can solve consensus for unbounded concurrency, and those that cannot, showing that consensus number alone is not sufficient to fully characterize object power.

2. RELATED WORK

Power of objects. The possibility that consensus power alone may not completely characterize the relation between shared objects was already raised by Herlihy with the introduction of the wait-free hierarchy [13]. This was proved for atomic registers by Herlihy, who showed a nondeterministic consensus power 1 object that cannot be implemented in a wait-free manner from registers for 2 processes [12]. Jayanti has considered whether an object hierarchy is *robust*, i.e., whether combining different "weak" objects enables implementations of stronger objects [15]. The exploration of consensus power 2 objects was initiated by Afek, Weisberger and Weisman in [3], where they defined the Common2 class of objects. Subsequently Common2 was shown to contain the stack object [2], various restricted versions of the FIFO queue [6–8,16,17], and the blurred history object [7]. In [3] Afek, Weisberger and Weisman sketched a wait-free `swap` implementation from 2-consensus objects, and the full construction appears in [21]. Gafni and Rajsbaum recently demonstrated a simple *recursive* `swap` implementation from 2-consensus [11], however it is not linearizable and is wait-free only under bounded concurrency.

Unbounded concurrency. The unbounded concurrency model was introduced by Merritt and Taubenfeld in [18], which focused on models with objects stronger than registers but not on wait-free computation. In [10], Gafni, Merritt and Taubenfeld explored unbounded concurrency wait-free computation using only read/write registers. They showed that increasing the allowed concurrency level leads to a weaker read/write computational model. Unbounded concurrency is the weakest level in this *concurrency hierarchy*, yet still many problems (such as snapshot and renaming) are solvable in this model. In [19], Merritt and Taubenfeld investigated consensus in the unbounded concurrency model. They focused on models where consensus is available as a base object, however it is constrained in the number of processes allowed to access it or the number of faults tolerated. Common2 was extended to the unbounded concurrency model in [2], where it was shown that except for the `swap` object all previously known Common2 objects had unbounded concurrency implementations from 2-consensus. Additional work on the unbounded concurrency model includes Aspnes, Shah and Shah's randomized unbounded concurrency consensus algorithms from registers [4] and Chockler and Malkhi's active disk paxos protocol for infinitely many processes [5].

3. PRELIMINARIES

System model. The system consists of a set of sequential processes and a set of atomic base objects that the processes use to implement high-level objects.

Objects. An *object* is defined by its *sequential specification*, which is a state machine consisting of a set of possible *states*, a set of *operations* used to transition between the states, and the possible *transitions* between the states. For every pair (s, op) of state and operation, there is a transition $T(s, op) = (s', r)$, such that invoking operation op when the object is in state s elicits the response r, and moves the object to state s'.

Implementations. An *implementation* of a high-level object O is a protocol specifying the base object operations that each process needs to perform when invoking the high-level operations of O in order to complete and return a response. The system's *state* consists of the state of all processes and base objects. An *execution* of the system is a (possibly infinite) sequence of *events*. Each event consists of a process invoking an operation on a base object and immediately receiving a response (since the base object is atomic), thereby moving the system to a new state.

Concurrency levels. As in [10,18], we assume the number of processes in the system is infinite. However, the *concurrency level* can be bounded: In an *n-bounded* model, at most n processes may be active concurrently. (This is essentially the standard n-process model where processes repeatedly arrive to invoke new operations [13]; we think of each new invocation as coming from a new process.) In a *bounded concurrency* model, the concurrency in every execution has some finite limit, however it is not known *a priori* and can differ between executions. Finally, in an *unbounded concurrency* model the concurrency may increase without bound as more and more processes join the execution.

Correctness. We require implementations to be wait-free and linearizable [14]. In a *wait-free* implementation the protocol guarantees that a process completes any operation in a finite number of its own steps, regardless of how other processes are scheduled in the execution. An implementation is *linearizable* if the high-level operations appear to take effect atomically during the invoking process' execution.

Object specifications. *Consensus* objects support a single `propose`(v) operation satisfying two properties: all `propose()` invocations return the same response (*agreement*), and this value is the input of some invocation (*validity*). We consider *binary consensus*, where only 0 or 1 can be proposed. A *c-consensus* object is a consensus object that can be accessed by at most c processes. A *swap* object holds a value (initially $\perp$) as its state, and supports a `swap`(v) operation. In state x, a `swap`(y) changes the state to y and returns x. A *test-and-set* (`T&S`) object supports a single `T&S` operation. Its state is a bit, initially `TRUE`. The `T&S` operation sets the bit to `FALSE` and returns the previous value of the bit. The first process to invoke `T&S` therefore receives the response `TRUE` and is said to *win* the `T&S`. All other `T&S` invocations *lose* and receive a response of `FALSE`. A *fetch-and-add* (`F&A`) object holds a natural number. A `F&A` operation takes a natural number x as input and adds x to the number stored in the `F&A` object, returning the original value in the response. A *fetch-and-inc* object is a `F&A` that is only used for increments and reads (by adding zero). An (unbounded concurrency) *snapshot* object [1, 10] holds an infinite array of registers. It supports two types of operations: $\texttt{update}_i(v)$, which writes v to cell i of the array, and `scan()`, which returns the (finite) prefix of the array written to before the `scan()`. That is, if the highest cell updated before a `scan()` is cell i, the `scan()` will return the sequence with the contents of cells $1, \ldots, i$.

4. SWAP IMPLEMENTATION

To gain some intuition on the difficulty of implementing `swap` from 2-consensus (even in the standard n-bounded model), consider that a process returning from a `swap` must know who is immediately before it in the linearization order so it can return its input. Yet the implementation must be such that the process cannot apply this knowledge transitively, or it would be able to determine the first process in the linearization order and solve consensus for arbitrary n. Obtaining this balance without compromising wait-freedom is not trivial. To demonstrate this, we sketch a natural idea for implementing `swap` that is flawed even in the n-bounded model. We then explain our new ideas for fixing the flaws in a way that is independent of the concurrency level.

Straw man algorithm. We use an array of `test-and-set` objects and a `fetch-and-inc` object. An arriving process obtains a unique array index using the `F&I` and then tries to *capture* that cell in the array. If it succeeds, it works its way backwards looking for another captured cell whose value it can return. Otherwise, it obtains another cell to compete in using the `F&I` and tries again. The linearization order is thus established based on the order in the array: the process that captures cell i is linearized after any process that captures a cell $j < i$. To avoid having two processes return the same value, process P_i must *block* any other process from capturing a cell in the range between cap_i (the cell P_i captures) and $ret_i = cap_j$ (the cell whose value P_i eventually returns). To do this, P_i searches for ret_i by competing in each cell $< cap_i$ in descending order until it loses some `T&S` (or falls off the bottom and returns $\perp$). Losing the `T&S` at cell c means that $c = cap_j$ for some P_j and P_i can return P_j's input, after it has ensured that no process can capture any cell between cap_j and cap_i.

Unfortunately, this straw man algorithm is not wait free. A process Q can *starve*: each time Q tries to capture some cell c, a new arriving process might capture a cell $c' > c$ and proceed to block c during its descent. Q must then try to capture a new cell larger than c', and this scenario may repeat forever[1]. To avoid this starvation, we would like to guarantee that only a finite number of competing processes can cause Q to lose at a cell and move to another one. This can be achieved if, beyond some point in the execution of an existing process Q, newly arriving processes will "move out of Q's way" and only try to capture cells *smaller* than the cell that Q is trying to capture. As a result, a newly arriving process P_i will never block Q while it descends from cap_i to ret_i, and so Q could lose only to the finite set of already active processes, achieving wait-freedom. To allow this our data structure must support ordering an *unbounded* number of cells between each two cells. This is because during the interval between Q failing to capture cell c_0 and competing in another cell c_1, an unbounded number of new processes may arrive with each one returning from a distinct cell c such that $c_0 < c < c_1$.

[1]This is possible with only two concurrent processes, so this problem is not unique to the unbounded concurrency setting.

Such a data structure was constructed in [3, 21] for the n-bounded model. The construction is a complicated infinite tree with unbounded node degree that guarantees wait-freedom only by inherently relying on having a fixed concurrency bound, n. In our new `swap` implementation we obtain a considerably simpler infinite tree structure that allows ordering an unbounded number of nodes between any two nodes. We exploit the structure of an *infinite binary tree* (Figure 1), where the **in-order** order on the nodes defines exactly the desired structure: each node is larger than its left subtree and smaller than its right subtree, so there is an unbounded number of nodes that can be ordered between any two nodes. Figure 1 depicts an example of this.

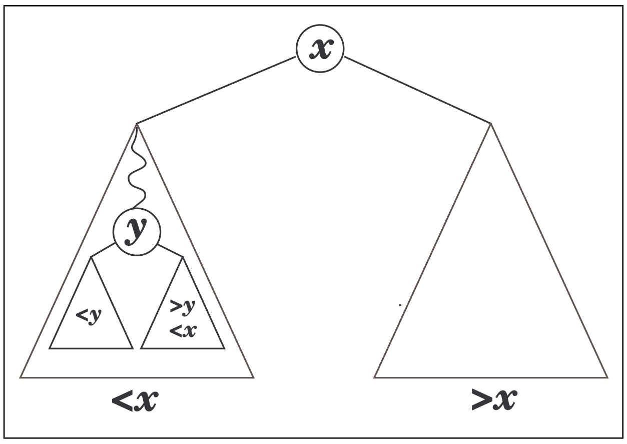
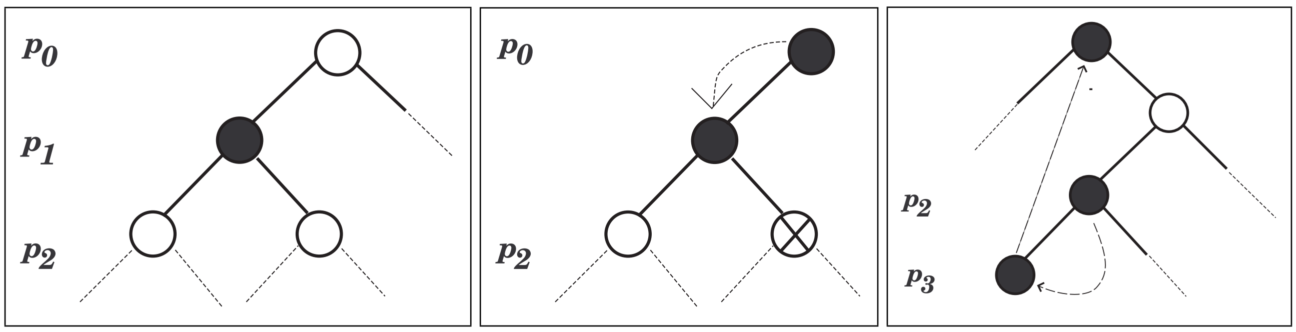


Figure 1: Infinite binary tree: Each triangle is an infinite binary tree. For each z in y's (infinite) right subtree, $y < z < x$.

Our algorithm (like the straw man algorithm) consists of a *capturing phase* and a *returning phase*. A process P_i initially obtains a unique depth (distance from the root) $depth_i$ using a `fetch-and-inc` object. It then enters the capturing phase, where it loops trying to capture the smallest node at $depth_i$ that is larger than any previously accessed node. P_i determines which nodes in the tree have been accessed by scanning a snapshot object in which each process announces the nodes it has accessed. Thus, after trying to capture some node v, P_i announces that it has accessed v in the snapshot. Once P_i captures a node, cap_i, it enters the returning phase, in which it looks for the largest captured node smaller than cap_i whose value it can return. Where can this node be? New processes observe that cap_i has been accessed and therefore move to the right in the tree and compete in nodes larger than cap_i. The only uncertainty is about existing processes: have they observed cap_i accessed and moved on to larger nodes, or are they about to capture some cell $< cap_i$? P_i must therefore *block* all nodes where another process might still be active. To do this, P_i reads the `fetch-and-inc` object and obtains $maxdepth_i$, the maximum possible depth a process in the capturing phase may still be in. It then iterates over all nodes with depth $\leq maxdepth_i$ and that are $< cap_i$ in descending order, attempting to block each one, until it loses some `T&S` (or wins them all and returns $\perp$). Notice that the number of nodes with depth $\leq maxdepth_i$ and that are $< cap_i$ is bounded (they are all contained in a finite binary tree). Figure 2 depicts an example execution.



(a) P_0, P_1 and P_2 arrive. P_1 captures its node and returns $\perp$.

(b) P_0 captures its node, blocks P_2 and returns P_1.

(c) P_2 moves to root's right subtree. New process P_3 enters this subtree and captures its node. The node P_3 captures is the smallest accessed node in the right subtree, so P_3 returns P_0. P_2 captures its node and returns P_3.

Figure 2: Example execution of Algorithm 1. Black nodes are captured. Crossed nodes are blocked.

To see how wait-freedom is obtained, consider process P_0 that obtained *depth* $= 0$. As long as P_0 is delayed and does not access the root, the root node will not appear in the snapshot. Every other process will therefore only "play" in the root's left subtree T_L. This is because such a process tries to capture the *smallest* node at its depth that is larger than any previously accessed node, and nodes in the left subtree are smaller than those in the right subtree. In addition, a process capturing a node in T_L never attempts to block the root node, since it only tries to block nodes smaller than its captured node. It is therefore guaranteed that once P_0 moves it captures the root node and announces the root has been accessed in the snapshot. From this point, T_L becomes "closed" to new arriving processes who all go into the root's right subtree T_R, since they need to capture a node larger than the root. Furthermore, a process Q that captures a node in T_R will never try to block a node in T_L. Because Q attempts to block nodes in descending order, it will always try the root and stop there before going into T_L. Thus, only P_0 and processes that already captured a node in T_L can block the nodes in T_L — this is a finite number of processes. Working recursively, similar intuition applies to P_1 with *depth* $= 1$. The only difference is that P_1 might fail to capture its first node if it is slower than P_0. In this case P_1 moves to T_R, where all other processes are "playing" in *P_1's left subtree*, T_{R_L}, and cannot block P_1. In general, a process P_i can lose a competition on a node only to a process with depth $< depth_i$, of which there is only a finite number, thus achieving wait-freedom. We capture this intuition more formally in Section 4.2.

4.1 Detailed algorithm description

We now explain our unbounded concurrency `swap` implementation (Algorithm 1) in detail. Each node in the infinite binary tree has two fields: a read/write register, *reg*, and a `test-and-set` object, *tst*. (We prove later in Lemma 3 that these `test-and-set` objects can be accessed by at most two processes, and can therefore simply be 2-consensus objects.) An unbounded concurrency snapshot object [10], called *accessed*, is used to record all the nodes that the processes try to capture. A process trying to capture nodes at

Algorithm 1 Wait-free, linearizable, unbounded concurrency `swap`

```
Shared variables:
    max_depth: unbounded concurrency fetch-and-add
               object initialized to 1
    tree: infinite binary tree, each node with the fields:
        reg: atomic register
        tst: test-and-set object
    accessed : unbounded concurrency snapshot object
procedure Swap(value)
1:      depth := F&A(max_depth, 1)      // Capture phase
        repeat
2:          cap := NextUnaccessedNodeAtDepth(depth)
3:          cap.reg := value
4:          win := T&S(cap.tst)
5:          update_depth(accessed, cap)
6:      until win
7:      max_depth := F&A(max_depth, 0) // Return phase
8:      ret := cap
        repeat
9:          ret := GetPrevNodeMaxDepth(ret, max_depth)
10:         if ret = $\perp$ then return $\perp$
11:     until !T&S(ret.tst)
12:     return ret.reg
end Swap

procedure NextUnaccessedNodeAtDepth(depth)
    S := $\{v \mid v \in tree$ with depth $depth$ such that
          $v > a\ \forall a \in$ scan(accessed)$\}$
    return the smallest node from S
end NextUnaccessedNodeAtDepth

procedure GetPrevNodeMaxDepth(cur, max_depth)
    S := $\{v \mid v \in tree$ with depth $\leq max\_depth$ and
          $v < cur\}$
    if $S = \phi$ then return $\perp$
    return the largest node from S
end GetPrevNodeMaxDepth
```

depth i of the tree posts the largest node it has accessed to cell i of the snapshot. Finally, an unbounded concurrency `fetch-and-inc` (`F&I`) object [2], *max_depth*, maintains the maximum depth accessed thus far. We use this `F&I` to simplify the presentation of the algorithm. Later we describe how to do without it.

Performing a `swap(`*value*`)` operation consists of two phases. The invoking process, P_i, begins the *capturing phase* (Lines 1-6) by obtaining a unique depth, $depth_i$, in which it tries to capture a node (Line 1). P_i then snapshots *accessed* and uses it to compute *cap*, the smallest node at $depth_i$ that is larger (in the in-order order) than all previously accessed nodes (Line 2). Then P tries to capture *cap* by writing *value* into its *reg* field and trying to win its *tst* field (Lines 3-4). Regardless of the outcome, P_i then marks *cap* as accessed in the *accessed* snapshot object (Line 5). This repeats until P_i captures some node denoted cap_i.

Once the capturing phase is over, P_i enters the *returning phase*. Here P_i first reads the `F&I` object *max_depth* to obtain $maxdepth_i$, the maximum depth accessible in the tree at this time (Line 7). From this point P_i tries to return from each of the accessible nodes starting from cap_i (Line 8). In each iteration P_i computes the next largest accessed node of depth $\leq depth_i$ that is smaller than cap_i (Line 9) and tries to return from it (or block it) by performing a `test-and-set` at that node (Line 11-12). If P_i wins all these `T&S`es then P_i is the first process in the linearization order and returns $\perp$ (Line 10).

Reducing reliance on Common2 objects. To gain insight on where the ability to perform consensus is crucial for implementing unbounded concurrency `swap`, we now show that the `F&I` object *max_depth* can be removed from Algorithm 1. The *max_depth* object serves two purposes: to obtain a unique depth for capturing nodes at the beginning of the capture phase, and to bound the maximum depth that an old process may try to capture a node in during the return phase. We can therefore replace it with a combination of a snapshot and the unbounded concurrency $(2k-1)$-renaming algorithm of [10] as follows. Each process starts its capture phase by writing its id into a new *ids* snapshot. It then renames itself and proceeds to use the new id as its unique depth (i.e., $depth_i$ is P_i's new name). In the returning phase, the process first scans the *ids* snapshot. The number of ids written in the snapshot cannot be smaller than the number of processes that completed the renaming. Hence, the maximum depth accessible to such processes (i.e., $maxdepth_i$) can be bounded by $2\,|ids| - 1$.

4.2 Wait-freedom proof

We proceed to prove that Algorithm 1 is wait-free. We denote by cap_i the node captured by process P_i (i.e., the value held by `cap` when winning the `T&S` in Line 4). We use ret_i to denote the node from which process P_i returns a value, or $\perp$ if P_i returns $\perp$. Finally, $depth_i$ and $maxdepth_i$ denote the values obtained by P_i from *max_depth* in Line 1 and Line 7 respectively.

The return phase of the algorithm is clearly wait-free, since during it a process P_i accesses a finite number of nodes with depth $\leq maxdepth_i$. The core of the wait-freedom proof is thus showing that the *capturing* phase is wait-free, i.e., that given enough steps a process always manages to capture a node. To show this we require the following:

LEMMA 1. *A process in the capturing phase cannot try to capture nodes in the right subtree of a node v before v itself has been recorded in accessed.*

PROOF. Consider towards a contradiction an execution of Algorithm 1 violating the lemma, and let the capture attempt of node v_r by some process P_i be the first such violating access. That is, v_r is in the right subtree of some node v at depth d, but an $\texttt{update}_d(v)$ has not been executed by the time P_i tries to capture v_r. Denote by v_l the largest node in v's left subtree at the same depth as v_r (i.e., $depth_i$). Figure 3 depicts this scenario.

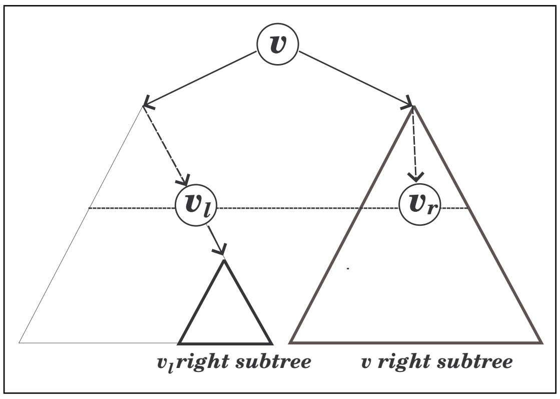


Figure 3: State of tree violating Lemma 1: P_i tries to capture v_r without v being written to *accessed* beforehand.

There are two cases to consider. Suppose P_i fails to capture v_l. Then some process P_j blocked v_l. Thus $cap_j > v_l$. In addition, since P_i eventually accesses v_r it follows that $cap_j < v_r$. Otherwise, because P_j (who tries to return from cells $< cap_j$ in descending order) reaches v_l it must block v_r first. But then after failing to capture v_l, P_i would observe cap_j in *accessed* (since P_j does no further updates to *accessed* after capturing a cell) and P_i would therefore not try to capture v_r at all, a contradiction. Therefore, $v_l < cap_j < v_r$. We now proceed to show that cap_j itself is captured before v_l is first written to *accessed*. By Lemma 3 below, only P_i can write v_l to *accessed*. This occurs after P_i fails to capture v_l. Thus, P_j is the first to access v_l, and P_j captures cap_j before that.

The remaining case is that P_i does not try to capture v_l. Thus P must observe some node $u \in$ *accessed* such that $v_l < u < v_r$ causing P to skip v_l, i.e. some process tried to capture u before v_l is written to *accessed*.

Either way, we find there is a node u that is accessed before both v and v_l are written to *accessed*, and satisfies $v_l < u < v_r$. Because v_l is the largest node in v's left subtree at depth $depth_i$, u must either be in v's right subtree or v_l's right subtree. This means the access to u violates the lemma, contradicting the assumption that the access to v_r is the first such event. $\square$

Lemma 1 implies wait-freedom of the capturing phase. This holds because a process can fail its capturing phase either due to losing at v, the right-most node of its level, or because it cannot find a node to access (i.e., it finds v smaller than some node in *accessed*). In any case this implies that some node in v's right subtree was accessed prior to v being accessed, contradicting Lemma 1. It remains to prove Lemma 3, which is implied by the following:

LEMMA 2. *If $cap_i > cap_j$ then $maxdepth_i > depth_j$.*

PROOF. Suppose this is false. Then P_i obtains $maxdepth_i$ from max_depth before P_j performs its initial F&I in its capture phase. Thus P_j in its capture phase observes cap_i in *accessed* and always tries to capture cells $> cap_i$, a contradiction. □

LEMMA 3. *For any node v, at most one process tries to capture it and at most one process tries to return from it.*

PROOF. By the code, it is clear that each process captures a node in a unique depth. Let us assume to the contrary that two processes, P_i and P_j, try to return the same node v. Say $cap_i > cap_j$. If $v > cap_j$, P_j will not try to return from it, a contradiction. Thus $v < cap_j < cap_i$. By Lemma 2, $maxdepth_i > depth_j$, and so, since P_i tries to return from v and P_i tries to return from cells in descending order, it must have tried to return from cap_j before that. As cap_j is captured by P_j, P_i must lose the `test-and-set` at cap_j, implying that P_i returns from cap_j and never accesses v, a contradiction. □

4.3 Linearizability proof

Here we prove that Algorithm 1 yields linearizable `swap` executions. This amounts to showing two things. First, that the `swap` specification is not violated. In other words, that the processes can be ordered so that the first process returns $\perp$, and subsequent processes each return the input value of the previous process. We refer to this as establishing a *chain* relation between the processes. The second task is showing that the chain relation respects the real-time ordering of processes: if P_i arrives after P_j returns, then P_i must be ordered after P_j.

We begin by proving the chain relationship. Given that Algorithm 1 is wait-free we consider only *finite* executions in which all of the processes finish their operations. This implies the theorem for unbounded concurrency as well, since any counter-example to linearizability occurs at a finite point in time, where our proof applies.

Definition 1. The relation $\rightsquigarrow$ is defined as follows: $P_i \rightsquigarrow P_j$ if P_i returns the value of P_j.

To show that the chain is well-formed we need the following lemmas:

LEMMA 4. *There is precisely one process that returns $\perp$.*

PROOF. Consider a process, P_{small} who captures the smallest node cap_{small}. This process will not be able to lose in any node and will return $\perp$ (Line 10). Now suppose two processes, P_i and P_j, both return $\perp$. Say $cap_i > cap_j$. From Lemma 2 we have $maxdepth_i > depth_j$ and so P_i must try and succeed to return from cap_j, otherwise P_j fails to capture it. This contradicts P_i returning $\perp$. Since P_j is able to capture that node, P_i is able to return cap_j and thus does not return $\perp$. □

LEMMA 5. *In a finite executions in which all of the processes finish their operations, there is exactly one process whose input is not returned by any process.*

PROOF. P_{big} will be the process who captured the largest node, cap_{big}. By the code, $\forall i\ ret_i < cap_i$. By definition of P_{big}, $\forall i\ cap_i \leq cap_{big}$. Therefore no process can return P_{big}'s value. □

LEMMA 6. *Each process returns precisely one value.*

LEMMA 7. *Each process's value can be returned by at most one process.*

PROOF. Process P_i can store its value in several nodes. However, in every node v but the last one it fails to capture the node, implying that another process P_v wins the the `test-and-set` at v. By Lemma 3, P_v must be a process trying to return from v who will not return from v (because it won the `test-and-set`). By Lemma 3, no process but P_v will try to return from v. Finally, for the final node that P_i captures, only one process can try to return that value (again by Lemma 3). □

Combining the above, we obtain:

LEMMA 8. *The relation $\rightsquigarrow$ is acyclic.*

PROOF. We know that $P_i \rightsquigarrow P_j$ implies that $ret_i = cap_j$ and $cap_i > ret_i$. From this we know that $P_i \rightsquigarrow P_j \Rightarrow cap_i > cap_j$. Thus a cycle $P_i \rightsquigarrow P_j \rightsquigarrow \cdots \rightsquigarrow P_k \rightsquigarrow P_i$ implies $cap_i > cap_j > \cdots > cap_k > cap_i$, which is impossible. □

We are now ready to prove:

THEOREM 9. *Algorithm 1 is linearizable.*

PROOF. Combing Lemmas 4-8 we have that the $\rightsquigarrow$ relation constructs a chain between the processes. We can now proceed to linearize the processes based on the total order established by the transitive closure of $\rightsquigarrow$. It remains only to show that this order respects the real-time order of process execution. Suppose this is not the case and some P_i that returns before P_j starts is ordered before P_j. Then $P_i \rightsquigarrow P_1 \rightsquigarrow P_2 \rightsquigarrow \cdots \rightsquigarrow P_j$. Therefore $cap_i > cap_j$. But if P_i terminates before P_j starts then when P_j starts its capture phase, cap_i is written to *accessed*, so P_j always tries to capture nodes $> cap_i$. This contradicts the assumption that $cap_i > cap_j$. □

5. UNBOUNDED CONCURRENCY APPLIED TO INFINITE CONSENSUS

In this section we explore consensus in the unbounded concurrency model. We first consider systems that have n-consensus base objects for all n (i.e., the number of processes allowed to access each consensus base object is finite). This model was first explored by Merritt and Taubenfeld in [19] and they showed that a system with n-consensus objects for all $n \geq 2$ can solve consensus in the face of *bounded concurrency*, where there are infinitely many processes, but the concurrency in every execution has some finite limit that is not known *a priori* and can differ between executions. Their consensus implementation fundamentally relies on the fact that *some* finite concurrency bound exists. Our first result in this section (Theorem 10 below) is that this is a fundamental limitation. Base objects capable of solving finite consensus for any finite set *cannot* solve unbounded concurrency consensus.

THEOREM 10. *There is no unbounded concurrency wait-free implementation of consensus from read/write registers and n-consensus objects for all n.*

PROOF. Assume that such an implementation exists. We construct an execution of the algorithm in which processes take an infinite number of steps but do not reach a decision, thereby contradicting wait-freedom. We say a state s of the system is P-*bivalent* for a set of processes P, if s can be extended with steps of processes in P to yield different decision values. Otherwise s is called P-*univalent*. A state is P-*critical* if it is P-bivalent and the next step of any process from P moves the system to a P-univalent state.

Our inductive construction follows. In each step we move the system from a P-critical state to a P'-critical state, for some $P' \supset P$. During this move all processes of P take a step. Thus, continuing this forever leads to an execution in which processes take an infinite number of steps without deciding and contradicts wait-freedom. The base case of the construction is simple: pick two processes p_1, p_2 with distinct inputs and execute them until a $\{p_1, p_2\}$-critical state is reached [9, 13].

Now assume we have a P-critical state s for some finite set P of size k. The standard FLP valency arguments applied to the k processes of P [9, 13] show that in s all processes of P are poised to access the same c-consensus object O for some $c \geq k$. However, consider what happens if we next schedule a process *not* from P (this would be the first step for such a process). While the next step of each process from P moves the system to a univalent state, we will show that there exists a new process whose next step from s will move the system to another bivalent state. Specifically, consider the set $P' = P \cup \{p_{k+1}, \ldots, p_c, p_{c+1}\}$ of size $c+1$, that is P with $c-k+1$ new processes added. Because $P \subset P'$, s is P'-bivalent. It cannot, however, be P'-critical as there can be at most c processes from P' poised to access O, leaving at least one process about to access a different object, which is impossible [9, 13]. The standard valency arguments show that from s there is an execution α, consisting of steps by processes from P', that leads to a P'-critical state s'. As before, in this state all processes from P' are poised to access a c'-consensus object O' for some $c' \geq c+1$. Hence all processes from P took a step in α (in s they were about to access a different object O). $\square$

This result implies that consensus is not interesting as a base object in the unbounded concurrency model: solving unbounded concurrency consensus cannot be done using n-consensus objects (for all n), so we would need a consensus object accessible to an unbounded number of processes, making the solution vacuous. Therefore, what can we say about *infinite power consensus objects*, those that can solve consensus for any n? Do all of them admit unbounded concurrency consensus implementations? Some infinite power consensus objects (such as `compare-and-swap` or Plotkin's `sticky-bit` [20]) have consensus algorithms that do not depend on the concurrency bound and work even in the face of unbounded concurrency. However, for other objects — like Jayanti's `weak-sticky` object [15] — the implementation depends on a known concurrency bound n. It is thus plausible that the unbounded concurrency model exposes a gap in the strength of infinite power consensus objects: some may not be strong enough to solve consensus in the unbounded concurrency model. Our main result in this section is that this is indeed the case.

THEOREM 11. *There exists an object O with infinite consensus number that cannot solve consensus in a wait-free manner in the unbounded concurrency model.*

In the following we prove Theorem 11. Section 5.1 introduces a new infinite power consensus object, the *iterator stack*. In Section 5.2 we show the iterator stack can solve consensus for any concurrency bound n, even if n is not known in advance. Yet it cannot solve unbounded concurrency consensus (Section 5.3).

5.1 The iterator stack

A state of the *iterator stack* consists of two (unbounded) sequences: V, a sequence of values that functions as a write-only stack (initially empty), and I, a sequence of non-negative integers called *iterators* (initially, I is all zeroes). The iterator stack provides two operations, `write()` and `read()`. A `write`(v) has two effects: (1) the value v is prepended to V (shifting previous values one place to the right), and (2) the first iterator with value 0 is set to 1 and its index is returned. The effect of a `read`(i) operation depends on whether iterator i points to an element of V. If $0 < I[i] \leq |V|$, then $V[I[i]]$ (the value iterator i points to) is returned. Otherwise, if iterator i is *uninitialized* ($I[i] = 0$) or is *invalid* ($I[i] > |V|$), then $\perp$ is returned. In any case, if $I[i] > 0$ then $I[i]$ is incremented (reading an invalid iterator does not stop it from being incremented). Figure 4 shows an example of an iterator's stack execution. Notice that an initialized iterator i can switch from returning $\perp$ to returning values from V if enough writes are performed in between the reads, increasing the size of V to the point where i is valid again.

Iterator stack examples
Each row consists of an operation applied to a given state, leading the object to the the state in the next row.

State	**Operation leading to next state**
$\langle V = [v_2, v_1], I = [3, 4, 0, 0, \ldots] \rangle$	`read`(3) **returns** $\perp$ (iterator 3 is uninitialized)
$\langle V = [v_2, v_1], I = [3, 4, 0, 0, \ldots] \rangle$	`read`(1) **returns** $\perp$ (iterator 1 is invalid)
$\langle V = [v_2, v_1], I = [4, 4, 0, 0, \ldots] \rangle$	`write`(x) **returns** 3
$\langle V = [x, v_2, v_1], I = [4, 4, 1, 0, \ldots] \rangle$	`write`(y) **returns** 4
$\langle V = [y, x, v_2, v_1], I = [4, 4, 1, 1, \ldots] \rangle$	`read`(1) **returns** v_1

Figure 4: Example state transitions of the iterator stack.

5.2 Iterator stack implements bounded concurrency consensus

Here we present Algorithm 2, a bounded concurrency consensus algorithm from registers and a single iterator stack. This shows that the iterator's stack consensus number is infinite. The idea behind Algorithm 2 is that participating processes write to the iterator stack, with the first process to write *winning* and its value becoming the *decision value*. To discover the decision value, a process traverses the iterator stack using the iterator it received when writing. When it reads the value $\perp$, it knows the previous read value was the first to be written, hence it is the decision value. However, notice that new processes can keep arriving and writing new values to the iterator stack, causing an iterating process to never reach the end of the sequence. This is due to the fact that while the *concurrency* is bounded, *arrivals* are not — so processes may continuously leave and enter the algorithm. To avoid such starvation, a process that determines the winner announces the winner in a *result* register. A new process will only write to the iterator stack if *result* is empty, adopting the value in *result* otherwise. This ensures the iterator stack is only written to a finite number of times.

Algorithm 2 Iterator stack bounded concurrency consensus protocol

```
    Shared variables:
        $result$ : atomic register, initialized to $\perp$
        $IteratorStack$ : iterator stack object
    Local variables:
        last, itr, cur : atomic registers, initialized to $\perp$

    procedure propose(value)
1:      if $result \neq \perp$ then return $result$
2:      itr := $IteratorStack$.write(value)
        repeat
3:          last := cur
4:          cur := $IteratorStack$.read(itr)
        until cur = $\perp$
5:      $result$ := last
6:      return $result$
    end propose
```

Theorem 12. *Algorithm 2 is a wait-free bounded concurrency consensus implementation.*

Proof. Validity is immediate. Similarly, we show agreement by proving that all writes to *result* write the same value. To see this, note that when a process first reads $\perp$, then the last value it read is the first value written to the iterator stack, and this is the value it writes to *result*. Wait-freedom follows from the fact that before *result* is written, only a finite number of processes can participate, due to the bound on concurrency. Therefore only a finite number of values can be written to the iterator stack, and so all iterations must terminate. □

5.3 Iterator stack cannot implement unbounded concurrency consensus

In this section we show that the iterator stack object is too weak to solve unbounded concurrency consensus. We assume towards a contradiction that such a consensus algorithm **A** exists. We derive the contradiction by adapting Fischer, Lynch and Paterson's *valency* argument [9] to the unbounded concurrency model. We call a state s of **A** *bivalent* if s can be extended to yield different decision values. Otherwise s is called *univalent.* All executions continuing from a univalent state s return the same value v, which is called s's *valency.* We also say that s is v-valent.

If we were in the n-bounded model, we could start with a bivalent state and execute each process until just before it moves the algorithm to a univalent state, ultimately reaching a bivalent state from which *any* next step leads to a univalent state. From this *bounded critical* state a contradiction can be derived by showing the different univalent states reachable from it are indistinguishable. However, this fails in the face of unbounded concurrency. There may always be a *new* process leading to a bivalent state, so we could end up with an execution in which infinitely many processes each take a single step, not violating wait-freedom. Instead, our notion of a *critical state* (below) is weaker, requiring two steps of certain processes to reach states with distinct valency.

Definition 2. A state s is *critical* if and only if s is bivalent, and there exist two processes, P and Q, such that: (1) from s, a single step of P leads to a p-valent state, (2) from s, a single step of Q followed by a single step of P leads to a q-valent state, (3) $p \neq q$.

Lemma 13. *A critical state of* **A** *exists.*

Proof. We construct an execution E leading to a critical state s_{crit}. We start with a bivalent state s_0. We let process P run solo from s_0 until the system reaches a state s_1, from which P's next step takes the system to a p-valent state. The state s_1 satisfies the following: (1) it is bivalent, (2) P moving at s_1 leads to a univalent state. Since s_1 is bivalent, there is an execution L_1 from s_1 that brings the system to a q-valent state, $q \neq p$. Notice that each step in L_1 but the last keeps the system in a bivalent state.

We now execute L_1 one step at a time until it is no longer the case that if P takes the next step, then the system moves to a p-valent state. This eventually happens, since the last step of L_1 leads to a q-valent state. When P's next step does not lead to a p-valent state, one of the following must hold: either (1) if P performs the next step the system moves to a bivalent state, or (2) if P performs the next step the system moves to a q-valent state. Note that condition (2) holds when L_1 ends at a q-valent state.

If (1) is the first condition to occur, we switch back to running P solo until we reach a state s_2 where, if P takes the next step, the system moves to some p_2-valent state (it could be that $p_2 \neq p$). The state s_2 satisfies the same two properties as s_1: both are bivalent and in both states P's next step brings the system to a univalent state. This means we can repeat the same procedure as done on s_1, executing a different extension L_2 leading from s_2 to a q_2-valent state ($q_2 \neq p_2$) one step at a time, until P's next step no longer leads to a p_2-valent state. If at this point P's next step moves the system to a bivalent state we run P solo again until its next step will take the system to a p_3-valent state, at which point we start scheduling an execution L_3, and so on. Eventually, during the execution of some L_i we must reach a state x where condition (2) holds, i.e., scheduling P next moves the system to a q_i-valent state. Otherwise, we would keep scheduling P forever without it completing, contradicting wait-freedom.

We denote each scheduling step of L_i by $l_1, \ldots, l_k$, so x is the state after the execution of $l_1, \ldots, l_k$. We claim the state s after the execution of $l_1, \ldots, l_{k-1}$ is a critical state. By construction, the next step of P at s moves the system to a p_i-valent state, for $p_i \neq q_i$. Therefore, the process Q performing the step leading from s to x cannot be P, since x is not p_i-valent. As condition (2) is true at x, P's next move from x leads to a q_i-valent state, and so a single step of Q at s (leading to x) followed by a single step of P leads to a q_i-valent state. The state s is thus indeed a critical state. □

We have established the existence of a critical state, s_{crit}. Let P be the process whose next step, o_P, at s_{crit} leads to a univalent state p-valent state, and Q the process whose next step, o_Q, at s_{crit}, followed by o_P, leads to a q-valent state, $q \neq p$. Let s_P be the p-valent state resulting from o_P, o_Q being scheduled at s_{crit}, and s_Q be the q-valent state reached by scheduling o_Q, o_P. We proceed to derive a contradiction by examining the different operations o_P and o_Q might be and ruling out each one. Due to our weaker definition of a critical state, s_P and s_Q will turn out to be distinguishable, differing in the state of an iterator stack object O. We will therefore prevent some *victim* process R from observing this difference by constantly adding new processes that write to O, preventing R from observing the difference in O's state. In doing so, these processes may themselves distinguish between s_P and s_Q and hence we do not allow them to perform any further operations on the shared memory. Here we use the unbounded concurrency requirement — we need an endless supply of new processes to induce an infinite execution of process R.

Let us first consider the types of operation o_P and o_Q can be (i.e., atomic register `read` or `write`, or iterator stack `read` or `write`). It is easy to see that o_P and o_Q may not *commute* since then s_P and s_Q would be indistinguishable states — all objects would have the same state in both. Therefore, o_P and o_Q must be operations on the same object O, and O cannot be an atomic register. To see this, recall that scheduling o_P at s_{crit} leads to a p-valent state, s_u. Thus o_P must be a write [13]. But then, s_Q and s_u would be indistinguishable to P, but with different valency — a contradiction. Additionally, o_P and o_Q cannot both be `read` operations on an iterator stack. This is because the only state change such a `read` affects is an increment of an iterator, a commutative operation. We now show that both operations must be iterator stack `write`s.

Lemma 14. *Both o_P and o_Q are* `write`*s to the same iterator stack O.*

Proof. Assume, w.l.o.g., that o_P is a `write` and o_Q is a read of iterator j. Let O's state at s_{crit} be

$$\langle V = [v_m, \dots, v_1], I = [i_1, i_2, \dots, i_m, 0, 0, \dots] \rangle .$$

o_P and o_Q commute in their effect on the shared memory state unless $j = m+1$, i.e., j is the iterator that the next `write` to O receives as a response. In this case, when scheduling o_Q first, iterator $m+1$ is uninitialized and so O's state does not change. Scheduling the `write` o_P after o_Q thus causes $I[m+1]$ to take the value 1. In contrast, scheduling o_Q after P's `write` causes an increment of $I[m+1]$ from 1 to 2. Thus, states s_P and s_Q differ only in the value of iterator $m+1$. Consider the solo executions of a new process R starting at each of these states, e_P and e_Q. Since s_P and s_Q are of different valency, R must distinguish between them by applying an operation that returns a different response in e_P than in e_Q. This can only be a `read` of O with iterator $m+1$. In e_Q (where $I[m+1]$ is initially 1) we stop R immediately after its first `read` of O with iterator $m+1$ (causing $I[m+1]$ to become 2). In e_P (where $I[m+1]$ is initially 1) we stop R immediately before its first such `read` of O. The state of all objects after these executions is identical, yet these are states with different valency, a contradiction. □

We are left only with the possibility that both o_P and o_Q are `write`s to the iterator stack O. To conclude the proof, we rule this out as well, reaching a contradiction.

Lemma 15. *Both o_P and o_Q cannot be* `write`*s to the same iterator stack O.*

Proof. Assume $o_P = \mathtt{write}(v_P)$ and $o_P = \mathtt{write}(v_Q)$. Let O's state at s_{crit} be

$$\langle V = [v_m, \dots, v_1], I = [i_1, i_2, \dots, i_m, 0, 0, \dots] \rangle .$$

Then in s_P, $V = [v_Q, v_P, v_m, \dots, v_1]$ and in s_Q it is $[v_P, v_Q, v_m, \dots, v_1]$. The state of I is identical. We introduce a new *victim* process R, which must be able to distinguish between s_P and s_Q, and construct executions in which R takes infinitely many steps without completing, contradicting wait-freedom of **A**. Consider two executions, e_P and e_Q, starting at s_P and s_Q respectively. We run R until it is about to perform a `read` of O using iterator i which will return v_P or v_Q, i.e., $I[i] = k$ and $V[k] = v_P$ or $V[k] = v_Q$. (Note the executions are identical up to these points.) Let T be the set of initialized iterators in O at this point. Note that, assuming no new value is written to O from here on, each $i \in T$ can be read only a finite number of times (perhaps 0) before it can no longer return v_P or v_Q. We thus repeat the following: introduce a new process and execute it until either (1) it `read`s from O using an iterator $i \in T$, or (2) it is about to `write` to O. No process can distinguish e_P from e_Q without applying an operation to O, so either (1) or (2) must occur. If only (1) occurs, then eventually all iterators in T become invalid and so any further `read` of O returns the same response in both executions, resulting in a contradiction. Thus we must eventually have two processes poised to `write` to O. We let each of them execute their `write` — this "shifts" V down two place, making $V[k]$ not point to v_P or v_Q. Notice that by construction, the writing processes cannot distinguish between e_P and e_Q since they did not read from O using an iterator from T, and so the values they write to O are the same in both executions. Finally, we schedule R's `read` of O using i, which returns the value in $V[k]$. Therefore in both e_P and e_Q this read returns the same value. We can indefinitely repeat the above process: go back to running R solo until it is about to read v_P or v_Q from O, then prevent it from doing this by using new processes. □

6. CONCLUSION

We have used unbounded concurrency to distinguish between different infinite power consensus objects. This extends Gafni, Merritt and Taubenfeld's work that used unbounded concurrency to present a hierarchy of concurrency levels within the class of read/write objects [10]. They show that there are read/write solvable problems that are solvable with a certain concurrency level but not with higher concurrency. Thus we have shown that also in the higher levels, the unbounded wait-free hierarchy does not look the same as the bounded wait-free hierarchy. Few of the remaining open questions are: Can these results be extended to sub-consensus objects? Can we provide a clean and simple characterization of unbounded concurrency infinite consensus objects?

Acknowledgments. We thank Eli Gafni and Gadi Taubenfeld for helpful discussions, and the anonymous referees for their comments.

7. REFERENCES

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40:873–890, September 1993.

[2] Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 218–227, New York, NY, USA, 2006. ACM.

[3] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 159–170, New York, NY, USA, 1993. ACM.

[4] James Aspnes, Gauri Shah, and Jatin Shah. Wait-free consensus with infinite arrivals. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, STOC '02, pages 524–533, New York, NY, USA, 2002. ACM.

[5] Gregory Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18:73–84, July 2005.

[6] Matei David. A single-enqueuer wait-free queue implementation. In *Proceedings of the 18th International Symposium on Distributed Computing (DISC'04)*, volume 3274 of *LNCS*, pages 132–143. Springer-Verlag, Jan 2004.

[7] Matei David, Alex Brodsky, and Faith Ellen Fich. Restricted stack implementations. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, volume 3724 of *LNCS*, pages 137–151. Springer-Verlag, Oct 2005.

[8] David Eisenstat. A two-enqueuer queue. arXiv:0805.0444v2 [cs.DC], 2009.

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

[10] Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 161–169, New York, NY, USA, 2001. ACM.

[11] Eli Gafni and Sergio Rajsbaum. Recursion in distributed computing. In Shlomi Dolev, Jorge Cobb, Michael Fischer, and Moti Yung, editors, *Stabilization, Safety, and Security of Distributed Systems (SSS 2010)*, volume 6366 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin / Heidelberg, 2010.

[12] Maurice Herlihy. Impossibility results for asynchronous pram (extended abstract). In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 327–336, New York, NY, USA, 1991. ACM.

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

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

[15] Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44:592–614, July 1997.

[16] Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, University of Toronto, 2001.

[17] D. Scott McCrickard. A study of wait-free hierarchies in concurrent systems. Technical Report GIT-CC-94-04, Georgia Institute of Technology, 1994.

[18] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'00)*, volume 1914 of *LNCS*, pages 164–178. Springer-Verlag, Oct 2000.

[19] Michael Merritt and Gadi Taubenfeld. Resilient consensus for infinitely many processes. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, volume 2848 of *LNCS*, pages 1–15. Springer-Verlag, Oct 2003.

[20] Serge A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, PODC '89, pages 159–175, New York, NY, USA, 1989. ACM.

[21] Hanan Weisman. Implementing shared memory overwriting objects. Master's thesis, Tel Aviv University, 1994.