On the Existence of Booster Types

Maurice Herlihy* and Eric Ruppert†
Computer Science Department
Brown University
Providence, RI 02912
herlihy@cs.brown.edu, ruppert@cs.brown.edu

Abstract

A data type's consensus number measures its power in asynchronous concurrent models of computation. We characterize the circumstances under which types of high consensus number can be constructed from types with lower consensus numbers, a process called boosting. In settings where boosting is impossible, we can reason about the synchronization power of objects in isolation.

We give a new and simple topological condition, called k-solo-connectivity*, sufficient to ensure that one-shot types cannot be boosted to consensus number k. (The booster type need not be one-shot; it can be arbitrary.) We also show that, for $k > 2$, any type that is not k-solo-connected can be boosted to consensus number k. For types that can be boosted, we establish an upper bound on the amount the consensus number can be increased. For finite types, these properties and bounds are computable. For deterministic one-shot types, the k-solo-connectivity property also exactly characterizes the types that have consensus number less than k.*

1. Introduction

A *data object* provides a set of *operations* that manipulate its *state*. For example, a queue object provides operations to enqueue and dequeue items in a sequence. An object's states and the meaning of its operations are determined by its *type* (in this example, FIFO queue). A *concurrent* object is one that is shared by multiple concurrent processes in a multiprocessor system. Concurrent objects are required to be *linearizable* [12]: operations may overlap in time, but they must appear to occur in a one-at-a-time order consistent with their real-time order.

An implementation of a concurrent data object is *wait-free* if it guarantees that any process will complete any operation in a finite number of steps, regardless of delays or failures of other processes. The wait-free property is important because modern concurrent and distributed systems are inherently asynchronous: processes are subject to long and unpredictable delays as a result of scheduling interrupts, page faults, communication delays, and failures. If a process fails or is interrupted while performing an operation on an object, other processes should be able to access that object [9, 20].

Under what circumstances can one construct a wait-free implementation of a particular type? Unlike sequential notions of computability, in which all reasonable models are essentially equivalent, concurrent computability is highly sensitive to the underlying model. For example, it is impossible to construct a wait-free two-process FIFO queue using atomic read/write registers, but it is possible using atomic compare-and-swap variables [10].

In the *consensus* problem [23], each process in a set of processes is given a private input value, and those processes must agree on one process's input in a finite number of steps, in the presence of halting failures and asynchrony. (The consensus problem for N processes is called N-consensus.) One way to measure a type's power is by its *consensus number* [10], the maximum number of processes that can achieve consensus using objects of this type together with read/write memory. More precisely, a set of types $\mathcal{T}$ has consensus number N if one can construct an N-consensus protocol using any number of objects of each of the types in $\mathcal{T}$ and any number of read/write registers, and there is no such protocol for $N + 1$ processes. Consensus numbers provide a good measure of synchronization power because N-consensus is *universal* for N-process systems: from a set of types that has consensus number N, one can construct a wait-free linearizable implementation of any N-process concurrent object [10].

In this paper, we address the following question: to what extent is the consensus number of a set of types $\mathcal{T}$ =

* Supported by NSF grant CCR-9912401.
† Supported by the Natural Sciences and Engineering Research Council of Canada.

$\{T_0, \ldots, T_\ell\}$ determined by the consensus numbers of the individual types T_i? The consensus number of $\mathcal{T}$, written $c(\mathcal{T})$, is at least $\max_{0 \le i \le \ell}(c(T_i))$, but can it be higher? This question, known as the *robustness* problem [13], is important because it determines whether we can reason about the synchronization power of types in isolation.

We say that a type T can be *boosted* to consensus number k, if there exists a type B, called the *booster type*, such that $c(B) < k$, but $c(B, T) \ge k$: B and T together have consensus number at least k. A boosting is *non-trivial* if $c(T) < k$. A class $\mathcal{C}$ of types is *robust* if no type T in $\mathcal{C}$ has a non-trivial booster type in $\mathcal{C}$.

Do there exist types that have non-trivial booster types? Surprisingly, perhaps, the answer is yes: there are examples of objects with low consensus number that can be combined to implement objects with higher consensus numbers [7, 14, 18, 21, 24, 26]. Nevertheless, the practical importance of these examples remains unclear. All modern architectures (to our knowledge) provide synchronization exclusively through read-modify-write operations (or something easily shown to be equivalent), and therefore belong to a class of types known to be robust [25].

In this paper, we focus on the class of *one-shot* types, encompassing objects that permit each process to invoke one operation. We show that any one-shot type that can be boosted must satisfy properties unlikely to occur in any realistic computer architecture. In particular, they must behave non-deterministically even when accessed by only one process at a time. Violations of robustness, while mathematically interesting, thus seem unlikely to arise in practice.

We give a new and simple topological condition, called *k-solo-connectivity*, sufficient to ensure that one-shot types cannot be boosted to consensus number k. (The booster type need not be one-shot; it can be arbitrary.) We also show that, for $k > 2$, any type that is not k-solo-connected can be boosted to consensus number k. For types that can be boosted, we establish an upper bound on the amount the consensus number can be increased. Both solo-connectivity and the upper bound are computable for finite types. For deterministic one-shot types, the k-solo-connectivity property exactly characterizes the types that have consensus number less than k. It follows from this that the class of deterministic one-shot types is robust.

The characterizations given in this paper are proved only for the class of one-shot types. However this class of types exhibits rather complex behaviour with respect to the robustness question. In fact, all of the non-robustness results cited above prove that a one-shot type can be boosted, usually by an object that allows more than one access per process. (See Section 3 for a description of one such object, equality negation.) Thus, the known non-robustness results fit into the framework used in this paper.

2. Model

Our model is similar to that of Jayanti [13]. A *concurrent object* is a data structure shared by concurrent processes. Each object has a *type*, characterized by a set of states, a set of operations, a set of ports, a set of result values, and a transition relation carrying each state, operation, and port triple to a non-empty set of result and state pairs. Objects can be initialized to any state.

An object exports a finite set of *ports* at which operations may be applied by concurrent processes. The bindings between processes and ports may be fixed in advanced (*hard-wired*), or determined dynamically (*soft-wired*). In this paper, we restrict our attention to one-shot objects with soft-wired ports. Each port of a *one-shot* object can be used only once by a unique process, and a process can use only one port of each object. We allow objects to be non-deterministic (the transition relation need not be a function), and non-oblivious (the transition relation may depend on the port). We assume that objects are linearizable [12].

The behaviour of a one-shot object can be described using notions taken from combinatorial topology (see [11, 22] for details). A *vertex* $\vec{v}$ is a point in a high-dimensional Euclidean space. Vertices $\vec{v}_0, \ldots, \vec{v}_n$ are *affinely independent* if $\vec{v}_1 - \vec{v}_0, \ldots, \vec{v}_n - \vec{v}_0$ are linearly independent. An *n-dimensional simplex* (or *n-simplex*) $S^n = (\vec{s}_0, \ldots, \vec{s}_n)$ is a set of $n+1$ affinely independent vertices. Geometrically, it can be thought of as the convex hull of its vertices. For example, a 0-simplex is a vertex, a 1-simplex a line segment, a 2-simplex a solid triangle, and a 3-simplex a solid tetrahedron. Simplex S is a (proper) *face* of simplex T if the vertices of S are a (proper) subset of the vertices of T. A *simplicial complex* (or complex) is a set of simplexes closed under containment and intersection. $\mathcal{L}$ is a *subcomplex* of $\mathcal{K}$ if every simplex of $\mathcal{L}$ is a simplex of $\mathcal{K}$. A *path* from vertex $\vec{s}_0$ to $\vec{s}_k$ in complex $\mathcal{K}$ is a sequence $\vec{s}_0, \ldots, \vec{s}_k$ such that each $(\vec{s}_i, \vec{s}_{i+i})$ is a 1-simplex of $\mathcal{K}$. A *component* of a complex is a maximal subcomplex of $\mathcal{K}$ containing vertices reachable from one another by a path.

An *invocation* $\vec{\imath} = \langle p, op \rangle$ is a vertex labelled with a port and an operation, and a *response* $\vec{r} = \langle p, res \rangle$ is a vertex labelled with a port and a result value. A set of $d+1$ invocations (or responses) for distinct ports forms an invocation (or response) d-simplex. Two simplexes are *matching* if they are labelled with the same set of ports.

Consider an $(n+1)$-process one-shot type T. Let $\mathcal{R}$ be the response complex for T, which contains all possible response simplexes. For each object state q, let $\mathcal{I}_q$ be the *invocation complex* which contains all possible invocation simplexes that can occur when the object is in state q. We define a function Δ_q that maps each simplex of $\mathcal{I}_q$ to a subcomplex of $\mathcal{R}$. For a d-simplex I of $\mathcal{I}_q$, the complex $\Delta_q(I)$ consists of a set of matching d-simplexes (and all of their faces). Each of the d-dimensional simplexes in $\Delta_q(I)$ describes a set of possible responses that a T object can return if it starts in state q and the operations of I are executed in any order. It follows from this definition that, for any invocation simplexes $I' \subset I$ in $\mathcal{I}_q$, each simplex of $\Delta_q(I')$ is also in $\Delta_q(I)$. When convenient, we omit explicit mention of q.

2.1. Related Work

The robustness question was first investigated by Jayanti [13, 14]. Chandra et al. [7] showed that the class of non-deterministic objects is not robust in the hard-wired model. Moran and Rappoport [21] showed that the class of deterministic objects is not robust in the restricted hard-wired model, where each process may use at most one port of each object. Peterson [24] also described booster types for this model. Schenk [26] proved that there is an oblivious one-shot object that can be boosted. (An object is called *oblivious* if the result of an operation cannot depend on the port where it was invoked.) However, his proof uses objects that can sometimes choose non-deterministically from an infinite number of possible state transitions when an operation is invoked. Lo and Hadzilacos [18] improved Schenk's result by showing that there is an oblivious one-shot object with bounded non-determinism that can be boosted.

Ruppert [25] showed that the class of objects that are equipped only with read-modify-write operations [16] is robust, as is the class of objects that provide operations that read the object's state without altering it. More generally, it has also been claimed that the class of all types with deterministic sequential specifications is robust [5], though the proof has not yet appeared. A contemporary paper making the same claim has since been withdrawn.

3. Solo-Connectivity

There is a strong connection between connectivity properties of an object's response complex and that object's ability to solve consensus. Consider the different possible interleavings of a set of operations, described by an invocation simplex I, performed on an object initialized to state q. If different schedules result in responses from different components of $\Delta_q(I)$, then processes performing these operations are implicitly agreeing on which component of $\Delta_q(I)$ will contain all the responses. In the following definition, we consider whether *solo* executions by individual processes can lead to different components of $\Delta_q(I)$, for some state q and invocation simplex I.

Definition 3.1 Let $1 < k \leq N$. A one-shot type T with N ports is *k-solo-connected* if, for every state q, there is a mapping σ_q from vertices of $\mathcal{I}_q$ to vertices of $\mathcal{R}$ with $\sigma_q(\vec{\imath}) \in \Delta_q(\vec{\imath})$ such that, for any invocation k-simplex $I = (\vec{\imath}_0, \ldots, \vec{\imath}_{k-1})$ in $\mathcal{I}_q$, the vertices $\sigma_q(\vec{\imath}_0), \ldots, \sigma_q(\vec{\imath}_{k-1})$ all lie in the same component of $\Delta_q(I)$.

Figure 1. Two-port fetch-and-inc complex.

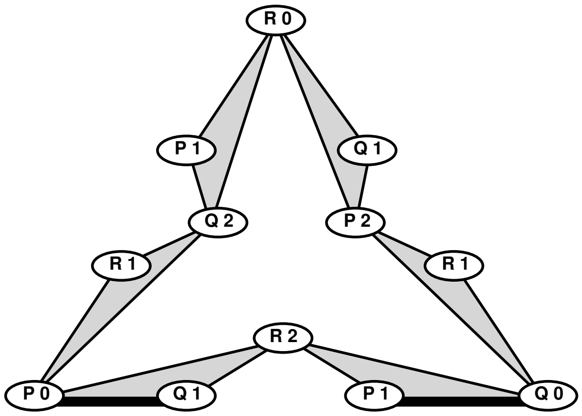

Figure 2. Three-port fetch-and-inc complex.

By convention, no type is 1-solo-connected. It follows from the definition that any k-solo-connected type is also $(k+1)$-solo-connected. To illustrate solo-connectivity, consider the following *fetch-and-inc* object. The object's state is given by an integer. The object provides one operation, fetch-and-inc, that atomically increments the object state and returns its prior value. Consider such an object whose state is initialized with 0. For an invocation $\vec{\imath}$ labelled by port P, $\Delta_0(\vec{\imath})$ consists of a single vertex labelled with P and 0, so there is only one possible choice for $\sigma_0(\vec{\imath})$. Figure 1 shows the response complex for two ports, denoted P and Q. A two-port fetch-and-inc object is not 2-solo-connected, because there is no path from $(P, 0)$ to $(Q, 0)$. Figure 2 shows the response complex for three ports, denoted P, Q, and R. (Vertices with the same label should be identified.) A three-port fetch-and-inc object is 3-solo-connected, because all responses corresponding to solo executions lie in the same component of this complex.

The *equality negation* object of Lo and Hadzilacos [18] is an interesting example of an object that is *not* solo-connected, even though its response complex is connected. This object has two ports, and provides the operation *negate*(v), where $v \in \{0, 1, 2\}$, which returns a binary value. Two *negate* invocations with distinct arguments must return the same value, while two invocations with identical arguments must return distinct values. The invocation and response complexes for this object are shown in Figure 3. The solid (dashed) 1-simplexes of the response complex represent legal responses for the solid (dashed, respec-

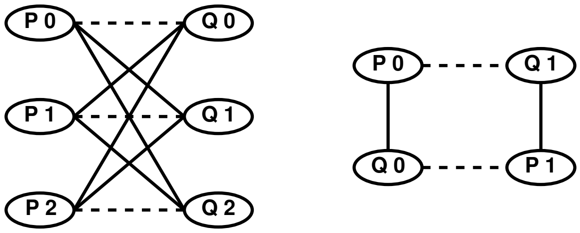

Figure 3. Invocation complex and response complex for equality negation.

tively) 1-simplexes of the invocation complex. We show that this type is not 2-solo-connected. Let $\vec{p}_i$ and $\vec{q}_j$ be the vertices $\langle P, i\rangle$ and $\langle Q, j\rangle$, and I_{ij} the invocation simplex $(\vec{p}_i, \vec{q}_j)$. Suppose we choose $\sigma(\vec{p}_0)$ to be $\vec{p}_0$. (The case where $\sigma(\vec{p}_0)$ is chosen to be $\vec{p}_1$ is symmetric.) We must then choose $\sigma(\vec{q}_1) = \vec{q}_0$, $\sigma(\vec{p}_1) = \vec{p}_1$, and $\sigma(\vec{q}_0) = \vec{q}_1$. So far, so good. Now $\sigma(\vec{p}_2)$ cannot be $\vec{p}_0$, because it is not connected to $\sigma(\vec{q}_0) = \vec{q}_1$ in $\Delta(I_{20})$. Similarly, $\sigma(\vec{p}_2)$ cannot be $\vec{p}_1$, because it is not connected to $\sigma(\vec{q}_1) = \vec{q}_0$ in $\Delta(I_{21})$. A similar analysis shows that the *weak agreement* object of Schenk [26] is not solo-connected. The equality negation and weak agreement objects can both be boosted.

We can now state the main result of this paper formally. The proofs of the two directions of this theorem are sketched in Sections 5 and 6.

Theorem 3.2 *A one-shot type T cannot be boosted to consensus number k if T is k-solo-connected. The converse is also true for $k > 2$.*

Corollary 3.3 *The consensus number of any k-solo-connected one-shot type T is less than k.*

Proof: Suppose not. Then registers would boost T to consensus number k (since registers have consensus number one [19]), contradicting Theorem 3.2. ■

The converse of this corollary is true for deterministic one-shot types, so we have the following characterization.

Theorem 3.4 *A deterministic one-shot type has consensus number at least k if and only if it is not k-solo-connected.*

Sketch of proof: One direction of the implication is given by Corollary 3.3. For the other, suppose T is a one-shot type that is not k-solo-connected. Then there is a state q and an invocation simplex $I = (\vec{\imath}_0, \ldots, \vec{\imath}_{k-1})$ such that the vertices $\Delta_q(\vec{\imath}_0), \ldots, \Delta_q(\vec{\imath}_{k-1})$ do not lie in the same component of $\Delta_q(I)$. (Since T is deterministic, $\Delta_q(\vec{\imath}_j)$ is a single vertex and therefore the unique possible choice for $\sigma_q(\vec{\imath}_j)$ in the Definition 3.1).

We describe how one T object, initialized to state q, and two registers can be used to solve team-restricted k-consensus, a restricted form of k-consensus where processes are divided, ahead of time, into two teams and all processes on the same team have the same input. It follows that T can also solve the general k-consensus problem [25]. Let team A consist of the processes P_j for which $\Delta_q(\vec{\imath}_j)$ lies in the component Z of $\Delta_q(I)$ that contains $\Delta_q(\vec{\imath}_0)$. Let team B consist of all other processes. Each process P_j first writes its input value into its team's register and then performs the invocation $\vec{\imath}_j$ on the T object. It outputs the value in team A's register if the response it receives is a vertex in Z, and the value in team B's register otherwise. ■

For a finite type T, it is computable whether T is k-solo-connected. Thus, one can decide whether a given finite deterministic one-shot type can solve k-process consensus, although this question is undecidable for arbitrary types [15].

Any M-ported object with $M < N$ can be treated as an N-ported object where the additional $N - M$ ports accept only a single operation that always gives the same result and causes no state change. Any such type is N-solo-connected via the result vertices for trivial operations. As a result, no one-shot type with $M < N$ ports can be boosted to consensus number N. An equivalent result, showing that M-consensus cannot be boosted to N-consensus, was given by Chandra et al. [7].

Theorem 3.5 *No deterministic one-shot type T can be boosted beyond its own consensus number.*

Proof: Suppose T has N ports and consensus number c. A standard valency argument [8] shows that $c \leq N$. If $c = N$, T cannot be boosted to a consensus number greater than c, by the argument in the preceding paragraph. If $c < N$, then T is $(c + 1)$-solo-connected, by Theorem 3.4. It follows from Theorem 3.2 that T cannot be boosted to consensus number $(c + 1)$. ■

This result is similar to the claim by Borowsky, Gafni and Afek [5] that class of all deterministic types is robust. Their claim applies to any deterministic type T (not just one-shot types), but covers only deterministic boosters B (whereas Theorem 3.5 also rules out non-deterministic boosters). Theorem 3.5 does not contradict the construction given by Moran and Rappoport [21], in which a 3-consensus object is boosted to consensus number N, because their proof relies in an essential way on the assumption that objects are hard-wired.

Given a non-deterministic type T that is not solo-connected, we can still bound how much its consensus number can be boosted. (For finite types, this bound will be computable.) A one-shot type T' is a *refinement* of one-shot type T if, for any state, the set of possible steps T' can take in response to any operation is a subset of the corresponding set of steps of T.

Theorem 3.6 *If T has a deterministic refinement T' with consensus number c', then no type can boost T beyond consensus number c'.*

4. Agreement Protocols

Borowsky et al. [4, 6] describe how a collection of processes can "simulate" the execution of a fault-tolerant, read/write protocol designed for a different number of processes. For brevity, we refer to the simulated processes as *threads*, and the simulating processes simply as processes. Each process simulates every thread, and processes run agreement algorithms to ensure that they are simulating the same execution. The heart of the simulation (henceforth called BG simulation) is a *safe agreement* protocol. Safe agreement is similar to consensus, except it is not wait-free. Instead, there is an *unsafe* region within the protocol during which a process will block the agreement protocol if it halts. The unsafe region of safe agreement contains a constant number of steps.

In the BG simulation, each process first uses safe agreement to fix each thread's input. Each process then visits each thread in round-robin order, attempting to execute the thread's next operation. Processes use safe agreement when simulating a read operation to ensure that every process produces the same response for the simulated step of the thread. A process never enters the unsafe region of a thread's safe agreement protocol if it is already in the unsafe region for another thread. This ensures that a process failure can prevent at most one thread from advancing.

4.1. Live Agreement

In Section 5, we introduce a new simulation that can handle objects other than registers. To do this, we use a new consensus protocol, called *live agreement*. The live agreement protocol for b processes uses only registers and objects that can solve $(b-1)$-process consensus. It is not wait-free, but any outputs produced are a correct solution to consensus. We first describe a simplified algorithm for the case $b = 2$ that uses only registers. Processes P and Q participate. Each process, say P, posts its preference to a shared array, and reads the other process's preference. If Q has not posted a preference, P decides its own preference. Otherwise, P repeatedly reads Q's preference. If they agree, then P decides that value. If they disagree, P sets its preference to agree with Q's, and continues re-reading Q's preference. Live agreement has no unsafe region: a process participating in a live agreement protocol on behalf of a thread is free to turn its attention to other threads at any time.

```
shared snapshot row[0,1,2,...]
shared (b-1)-consensus con[0,1,2,...]

value liveAgree (value input)
  p = input
  i = 0
  output = null
  while output == null
    write p in row[i].prefer
    read row[i]
    if fewer than b processes have written
      in row[i] then
      p = result of proposing p to con[i]
      write p in row[i].result
      read row[i]
      if fewer than b processes have written
        in row[i] then output = p
      end if
    elsif some process has written a value v
      in row[i].result then p = v
    end if
    i = i+1
  end while
  return output
end liveAgree
```

Figure 4. Live agreement for b processes.

Live agreement complements safe agreement in the following sense. Safe agreement can be thwarted if one process halts in its unsafe region, but will succeed if all processes take enough steps. By contrast, live agreement can be thwarted if the processes take the same number of steps (continually exchanging preferences), but will succeed if one process runs slower than the others. Suppose processes P and Q are both trying to solve two (separate) consensus problems in parallel. If they use live agreement for one and safe agreement for the other, then one or the other is guaranteed to eventually succeed.

We now sketch the live agreement algorithm for more than two processes. A pseudocode description is given in Figure 4. Each snapshot object row[i] has a single-writer, multi-reader register for each process. Each register has two fields, pref and result. The entire snapshot object can be read atomically.

Each process executes a loop. During the ith iteration, it writes its preferred value to a fresh area of shared memory and then takes an instantaneous snapshot of the values written by all of the processes. (Such a snapshot operation can be implemented from ordinary registers [1, 2, 3].) Processes that see fewer than b values written submit their own preferred values to a $(b-1)$-consensus object, write the response r to shared memory and take a snapshot again. If, during the second snapshot, a process sees that the number of processes that have started iteration i is still less than b, it halts and outputs r. Otherwise, it continues to the next iteration using r as its preferred value. Any process that sees, during its first snapshot, that all b processes have started iteration i checks if the response r has been written by any process. If so, it changes its preferred value to r.

Termination is guaranteed whenever one process can complete an iteration before another process starts the iteration, but the algorithm may not terminate if all processes are running at approximately the same speed. If any process terminates during iteration i, all others must also terminate, either during iteration i or $i+1$. It can be argued that any process that begins iteration $i+1$ must do so with r as its preferred value. It follows that every process outputs r.

4.2. Near Agreement

We also introduce the following variation of live agreement, called *near agreement*. A simplicial complex $\mathcal{K}$ is stored in shared memory. Each process is given a vertex of $\mathcal{K}$ as input, and the processes must output adjacent vertices of $\mathcal{K}$. (By *adjacent*, we mean the outputs are either equal or lie on a common simplex.) While the protocol is in progress, however, new simplexes may be added to $\mathcal{K}$ a finite number of times. (More precisely, the representation of $\mathcal{K}$ stored in read/write memory may be updated by concurrent threads, and rescanned by the processes.) As long as the input vertices lie in distinct components of $\mathcal{K}$, near agreement behaves like live agreement: the protocol may fail to converge as long as all processes run at approximately the same speed, but will converge if one process takes fewer steps than the others. If, however, the input vertices lie in the same component, then the protocol becomes wait-free: each non-faulty process will eventually decide on a correct value.

We now describe two-process near agreement in more detail. Each process has a *preference* stored in a shared array. While the preferences lie in distinct components of the complex $\mathcal{K}$, each process continually flips its preference to the other's until it observes that both preferences agree. If a process observes that the two preferences are different but lie in the same component of $\mathcal{K}$, it computes a path between them. (This path is determined by the initial preferences and remains fixed throughout the near agreement algorithm.) The process then begins successively adjusting its preference to move it one step along the path towards the preference of the other process until it observes that the two preferences are adjacent. This procedure will terminate, provided the initial preferences lie in the same component of the complex at some time.

The near agreement algorithm can be generalized to more than two processes in much the same way as the live agreement protocol. (See Figure 5. Each row[i] object has the same structure as in the live agreement protocol.) To ensure that preferences converge along a single path once the complex becomes connected, $b-1$ of the processes use additional $(b-1)$-consensus objects to agree on a common preference at each iteration.

```
shared snapshot row[0,1,2,...]
shared (b-1)-consensus con1[0,1,2,...]
shared (b-1)-consensus con2[0,1,2,...]
shared (b-1)-consensus initial

value nearAgree (Vertex input)
  // For process #1:
  p = input
  // For the other b-1 processes:
  p = result of proposing input to init

  i = 0
  output = null
  while output == null
    write p in row[i].prefer
    read row[i] (and the simplicial complex)
    if fewer than b processes have written
      in row[i] then
      new = result of proposing p to con1[i]
      write new in row[i].result
      read row[i]
      if fewer than b processes have written
        in row[i] then output = new
      end if
    elsif all vertices written in
      row[i].prefer lie on a common simplex
      then output = p
    elsif some process has written a value v
      in row[i].result then new = v
    elsif complex is of dimension k then
      new = neighbour of p on path between
        the two vertices in row[i].prefer
    end if

    // For process #1:
    p = new
    // For the other b-1 processes:
    p = result of submitting new to con2[i]

    i = i+1
  end while
  return output
end nearAgree
```

Figure 5. Near agreement for b processes.

5. Simulation

We now describe the simulation used to prove Theorem 3.2. Suppose there exists a wait-free k-process consensus protocol Π using a one-shot k-solo-connected type T and an arbitrary type B. Assuming that $c(B) \geq b-1$, where $2 \leq b \leq k$, we shall describe how b processes can simulate Π using only objects of type B and read/write registers. Processes execute safe agreement, using their own input values, to agree on the input to be used for each thread. When any thread terminates, processes output the value decided by that thread. This simulation yields a b-process consensus protocol, showing that $c(B) \geq b$. Thus, by induction, we have $c(B) \geq k$, so B does not boost T to consensus number k.

A key technical difference between our simulation technique and the BG simulation is that in BG-simulation, every process simulates every step of every object. Our simulation, however, must treat the B objects as black boxes. Processes cannot simulate B operations and then agree on the outcomes because it may not be possible to inspect or copy a B object's state without changing it.

5.1. Two Processes, k Threads

We first consider the case where there are two processes (i.e. $b=2$) and more than two threads (i.e. $k>2$). A thread encounters three types of objects: read/write registers, objects of booster type B, and objects of one-shot k-solo-connected type T. Read/write registers are simulated using safe agreement exactly as in the BG simulation. A process that fails in the middle of simulating a register operation can block at most one thread.

5.1.1 Booster Objects

To simulate a thread's call to a booster object operation, the processes undertake a safe agreement protocol, called the *booster entry protocol*. Each process uses its own process id as its input to safe agreement. The winner is responsible for actually calling the booster object's operation, and then storing the response in a shared register. (Here we exploit the assumption that object ports are soft-wired.) The interval between the read step in which the winner observes that it has won the entry protocol, and the write step in which it writes the outcome to a shared variable is an unsafe region, and that thread may not simulate any other threads in this interval.

Care is needed to ensure that a process failure does not block too many threads. Consider the following scenario. Suppose P and Q start a booster entry protocol for thread θ, P exits the unsafe region of safe agreement, but Q pauses in the unsafe region, blocking agreement. As soon as P turns its attention to thread θ', Q wakes up and finishes its unsafe region, losing to P. This leaves θ blocked until P executes the booster operation. If P then fails in its unsafe region for θ', both θ and θ' will be blocked by a single process failure.

If P stops simulating a thread during a booster entry protocol in order to move on to another thread, then Q must be in the entry protocol's unsafe region. P then considers that thread to have high priority. Each time a process stops simulating a low priority thread, it checks each high priority thread to determine whether the entry protocol has terminated. If so, then P lowers the thread's priority. If P won, then P is in the unsafe region in which it must execute the booster object operation and store the result before it can simulate any other threads.

This structure ensures that a single process failure can block at most two threads, as in the scenario above. If P has two high-priority threads, then the entry protocol of the earlier one must have terminated (since Q cannot block both at once), and P will immediately execute the B operation and record its result. Since there are more than two threads by hypothesis, a single process failure cannot block all threads.

5.1.2 T Objects

We associate *invocation* and *response* registers with each port of each T object. A thread calls a T operation by writing the invocation to the invocation register, and blocking until the response appears in the response register. An operation is *pending* if the invocation register has been written, but its response register has not been written. A thread is blocked while it has a pending operation.

Processes alternate between simulating threads and simulating T objects as described in Section 5.1.3, below. The routine for simulating a T object is shown in Figure 6. Let q be the initial state of the T object. At any time, the object's current set of invocations defines a *current invocation simplex* I, and a *current response complex* $\Delta_q(I)$. (We describe, below, how computations on these complexes are carried out.) Note that both the current invocation simplex and current response complex are enlarged periodically as new invocations arrive. However, at most k such changes occur. When invocations first appear, the processes conduct a near agreement protocol on the current response complex. If process P observes invocation $\vec{\imath}$, it takes $\sigma_q(\vec{\imath})$ as input to the near agreement, and writes the output to a shared array *start*$[P]$. (The function σ_q is defined in Definition 3.1.) Process P then stores in *prefer*$[0][P]$ some linearization of the pending operations that is consistent with the responses it observes in the *start* array. The near agreement specification ensures that *prefer*$[0][P]$ and *prefer*$[0][Q]$ agree on the response for at least one operation.

Process P then undertakes a sequence of live agreements. For each round i, $0 \leq i < k$, P uses as its input a linearization of the operations invoked so far that is consistent with the responses that have been agreed upon in previous rounds. Before the live agreement protocol, if P observes that the linearizations in *prefer*$[i][P]$ and *prefer*$[i][Q]$ both

```
shared Vertex start[b];
shared linearization prefer[k][b];

void simulateT ()
  wait while no invocations present
  s = any invocation
  start[P] = nearAgree(sigma(s))
  take snapshot of start array and of
    invocation registers
  prefer[0][P] = a linearization of the
    operations invoked so far consistent
    with responses seen in start array
  for i = 0..k-1
    run live agreement using prefer[i][P] as
      input.  (If there is a response that
      is the same in prefer[i][j] for all j,
      write it to response register)
    write responses from result of live
      agreement to response registers
    wait while no operations pending
    take snapshot of invocation and response
      registers
    prefer[i+1][P] = some extension of
      agreed-upon linearization, including
      all operations pending in the snapshot
  end for
end simulateT
```

Figure 6. Code for process P to simulate a T object.

return the same responses to some operations, P immediately fills in those responses (unblocking those threads). Since *prefer*$[0][P]$ and *prefer*$[0][Q]$ have the same response for at least one operation, at least one thread will be unblocked as soon as the initial near agreement succeeds. If the live agreement succeeds, P fills in the remaining responses, and waits for more invocations to appear before going on to the next round.

The set of all T objects used by the simulated protocol can be linearly ordered. At any time, some T objects may have pending invocations. P simulates the lowest T object in the ordering with a pending T operation, and Q simulates the highest. This division of labour ensures that if two distinct T objects have pending invocations, each will be simulated by a distinct process, and at least one thread with a pending invocation will receive a response.

During the course of the simulation of T objects, processes must execute near agreement on a response complex. It is not necessary to represent this complex explicitly in memory. Instead, it is defined by the set of invocations pending at the object, which are stored in shared registers. As long as there are fewer than k pending invocations, no computations involving the complex are required. However, once k invocations appear, processes may have to compute the a path between a given pair of vertices. This can be done by executing a search of the complex (constructing vertices as necessary).

5.1.3 Scheduling

Each process executes the following loop. First, it checks for pending T operations. If there are pending operations, P simulates the lowest-ordered T object, while Q simulates the highest. The processes then simulate each thread in round-robin order, just as in BG simulation, except that between each thread, the process checks whether any booster entry protocols of high-priority threads have become unblocked. If so, the process lowers that thread's priority. If it won, it executes the operation. If a low-priority thread becomes blocked during a booster entry protocol, the process assigns that thread a high priority.

5.1.4 Termination

We now outline an argument that this simulation is wait-free. Recall how a process terminates the protocol: if any thread finishes, the process can halt with the same decision value as the thread. If no process finishes in an execution, then no simulated thread finishes either. Suppose, by way of contradiction, that there is an infinite execution in which no thread finishes the consensus protocol. We consider two cases.

First, suppose both P and Q take an infinite number of steps. Then no thread can be blocked in the middle of a safe agreement protocol, or waiting for a booster operation to complete. All threads must be blocked at T operations. If the operations are on distinct T objects, then (as noted above), one must complete. If the operations are pending on a single T object, then every port has an invocation, and the preferred vertex of each process lies in the same component of the current response complex. The near agreement protocol will therefore terminate after a finite number of steps.

Second, suppose P alone takes an infinite number of steps. Then P must eventually finish any live agreement or near agreement protocol, so all threads must be blocked at read/write operations or booster operations. If Q halted in the unsafe region of a safe agreement module, then (as noted above) it can block at most two threads. Since there are at least three threads, at least one thread must finish the protocol.

5.2. Two Processes, Two Threads

If there are two processes and two threads (i.e. $b = k = 2$), then each process directly executes a single thread.

There is no need for a booster entry protocol or a BG simulation of registers, since the two processes can simply access B objects and registers directly. Objects of type T are simulated as described above. Each process alternates between simulating its assigned thread and T objects that have pending invocations.

If one process fails, every invocation to a T object will eventually receive a response, since live and near agreement terminate in this case. Thus, the thread simulated by the non-faulty process will complete its execution. If both processes continue to take steps, the simulation terminates for the reasons outlined in the previous section.

5.3. b Processes, k Threads

Assume inductively that B has consensus number at least $b-1$, where $2 < b \leq k$. We now show how to simulate the k-thread consensus protocol Π that uses B objects, T objects and registers in a system of b processes that can use only B objects and registers. This will show that the consensus number of B is at least b.

Let C be an object that solves $(b-1)$-process consensus constructed from B objects and registers. For accesses to read/write registers, we use BG simulation as before. For each access to a B object, we can construct a 2-level tree of C objects. To execute the B operation, each process joins the first level of consensus, and the winner proceeds to the second level. The top-level winner executes the B operation and writes the response into a register. (Here, too, we exploit soft-wired ports.) Failure of a single process can block at most one thread, since the thread is blocked only if the process fails in the unsafe region between the time it wins at the top level of consensus and the time it writes the response to the B operation into shared memory.

To simulate a T object, we use the same procedure as in the two-process case, replacing the two-process near and live agreement protocols by their counterparts for b processes that can be built from C objects and registers. If threads are waiting for responses at several T objects, process 1 simulates the lowest numbered T object where threads are waiting and the remaining processes simulate the highest numbered T object.

We now argue that the simulation terminates by considering two cases.

First, if some process fails, no thread can be blocked at a T object, since the live and near agreement protocols always terminate whenever a process fails. Furthermore, a halted process can block at most one thread at a B object or register, since the process only enters the unsafe region of one thread at a time. Thus, some thread (and hence all non-faulty processes) will terminate.

On the other hand, if every process continues to take steps, no thread can get stuck at a B object or register. Threads cannot be blocked indefinitely at different T objects, as in the two-process case. If all threads are blocked at the same T object, the response complex of that object becomes k-dimensional and connected, and the near agreement protocol will produce a response for at least one of the threads, thereby unblocking it. Thus, threads cannot be blocked indefinitely at the same T object, so some thread will always continue to make progress, and all processes will eventually complete the simulation.

This inductive proof establishes that $c(B) \geq k$, completing the proof of one direction of Theorem 3.2: it is impossible for B to boost a one-shot, k-solo-connected type T to consensus number k.

6. Boosters Exist

Let T be an N-ported one-shot type. We have seen that, if T is k-solo-connected, it cannot be boosted to consensus number k. In this section we show the converse holds for $k > 2$.

The proof technique used here was originated by Lo [17], and was later used to prove that two one-shot types could be boosted [18, 26]. However, those proofs were intimately tied to properties of the individual types they studied. Here, the technique is generalized to handle any boostable type.

Suppose T is not k-solo-connected. Let q be a state of T for which there exists no σ_q satisfying the conditions of Definition 3.1. We design a non-deterministic type B that will solve k-process consensus, provided the processes can "convince" the object that they can implement a T object, initially in state q. The B object is equipped with *propose* and *extract* operations. The *propose*(v, p) operation takes a natural number v (which can be interpreted as an input for the consensus problem) and a port p as arguments and returns an arbitrary invocation for port p of a T object. This invocation is called a *challenge* invocation. If more than k *propose* operations are performed, or if two are done with the same port argument, the B object returns arbitrary responses to all subsequent operations.

If the processes can cooperatively compute consistent responses for the challenge invocations, they can use them as arguments to the *extract* operation, and the B object will reveal a solution to consensus. The *extract*$(\vec{r})$ operation takes a response vertex for T as an argument and outputs a natural number. When an *extract*$(\vec{r})$ operation is done, the B object checks whether $\vec{r}$ is acceptable by seeing whether the set of arguments to the *extract* operations performed so far are distinct and form a simplex in $\Delta_q(I)$, where I is the simplex of challenge invocations that the B object has previously issued. (If the B object has chosen a set of challenge invocations that do not form a legal invocation simplex I for the object T, then any responses are deemed acceptable.) If $\vec{r}$ is acceptable, the B object returns the first value v that was proposed to it. Otherwise the B object becomes "upset". Once the object is upset, it returns arbitrary responses to all future operations.

If k processes have access to both a B object and a T object, they can use the T object to solve the challenge posed by the B object, and thereby solve consensus. Registers are used to avoid passing illegal challenges to the T object. Thus we have the following proposition.

Proposition 6.1 *There is a k-process consensus protocol that uses one object of type T, one object of type B and registers.*

We can now prove the second half of Theorem 3.2.

Theorem 6.2 *If T is a one-shot N-ported type that is not k-solo-connected (where $2 < k \leq N$) then T can be boosted to consensus number k.*

Sketch of proof: We must show that $c(B) < k$. Assume that $c(B) \geq k$ to derive a contradiction. We first define a notion of an implementation of T that has weaker correctness properties than usual. Consider an N-process protocol, where process p has, as its input, an invocation for port p of type T. We call the protocol a *permuted k-implementation of T* (for the initial state q) if, for any execution where at most k processes take steps and the inputs of those processes are distinct and form a legal invocation simplex I, all of the non-faulty processes eventually output distinct responses that form a simplex in $\Delta_q(I)$. Since B has consensus number k, it can implement any object in a system of k processes [10]. It follows that there is a permuted k-implementation of T that uses registers, fetch-and-inc objects and a finite number of B objects. We shall construct a permuted k-implementation of T that uses fewer objects of type B. By induction, it follows that there is an implementation using only registers and fetch-and-inc objects, which we shall see is impossible.

Consider an execution of Π where at most k processes take steps. Let I_0 be the simplex of inputs to processes. A configuration of Π consists of the state of all shared objects and the internal state of all processes at some point in an execution. A configuration C is called univalent if, in every execution continuing from C, the outputs are always in the same connected component of $\Delta_q(I_0)$ and multivalent if there are two executions that lead to outputs in different components.

A number of facts about Π can be proved using valency arguments [8]. The fact that T is not k-solo-connected can be used to show that Π has a multivalent configuration. Because Π is wait-free, there is a multi-valent configuration C where one step by any process will move the system into a univalent configuration. In this critical configuration C, every process is poised to access the same B object, O. (This is where the assumption $k > 2$ is required.) Thus, there is no permuted k-implementation of T from registers and fetch-and-inc objects.

Further valency arguments show that, in C, every process is poised to perform a *propose* operation on the object O, which has not been accessed before C. Because C is critical, the scheduler's choice of the next step determines which component of $\Delta_q(I_0)$ will contain the outputs of the processes. Every process running Π must therefore be able to discover some information about which *propose* operation was the first to access O after C. Since the *propose* operations return arbitrary responses, they provide no information about other *propose* operations that have taken place. So processes must access O a second time, using an *extract* operation. Consider the executions from C where processes halt immediately after accessing O a second time. In such executions, the only way for a process to obtain information about O's history is to use an *extract*($\vec{r}$) operation with an acceptable response $\vec{r}$ (since an unacceptable response upsets the object).

We define a new permuted k-implementation Π' of T by having processes simulate the portion of Π between the *propose* operation on O and the later *extract* operation on O. The protocol Π' uses the same set of objects as Π, except for O. (The processes also use an additional fetch-and-inc object to decide which process of Π to simulate.) This is where we use the fact that the implementation is allowed to permute the responses to challenge invocations: since objects are soft-wired, we have no way of ensuring that the response vertex used by a process for an *extract* operation in Π matches the challenge issued to that process. ■

Acknowledgement

We thank the referees for their helpful comments.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sept. 1993.

[2] J. H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, Apr. 1993.

[3] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, 1990.

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

[5] E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense! In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 363–372, Aug. 1994.

[6] E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. Technical Report TM-573, Laboratory for Computer Science, Massachusetts Institute of Technology, Dec. 1997.

[7] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of the wait-free hierarchies. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 334–343, 1994.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[9] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *2nd Symposium on Operating Systems Design and Implementation*, pages 123–136, Oct. 1996.

[10] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, Jan. 1991.

[11] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, Nov. 1999.

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

[13] P. Jayanti. Robust wait-free hierarchies. *J. ACM*, 44(4):592–614, July 1997.

[14] P. Jayanti. Solvability of consensus: Composition breaks down for nondeterministic types. *SIAM J. Comput.*, 28(3):782–797, Sept. 1998.

[15] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 69–84, 1992.

[16] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, Oct. 1988.

[17] W.-K. Lo. More on t-resilience vs. wait-freedom. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 110–119, 1995.

[18] W.-K. Lo and V. Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000.

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

[20] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University Computer Science Department, 1991.

[21] S. Moran and L. Rappoport. On the robustness of h_m^r. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, volume 1151 of *LNCS*, pages 344–361, 1996.

[22] J. Munkres. *Elements Of Algebraic Topology*. Addison Wesley, Reading MA, 1984.

[23] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[24] G. L. Peterson. Properties of a family of booster types. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, page 281, 1999.

[25] E. Ruppert. Determining consensus numbers. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 93–99, 1997. To appear in *SIAM J. Comput.*

[26] E. Schenk. The consensus hierarchy is not robust. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, page 279, 1997.