

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

April 25, 2000

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 of lower consensus number, a process called boosting. The circumstances under which types can be boosted determines the degree to which we can reason about the synchronization power of objects in isolation. We give a new and simple topological condition, called *solo-connectivity*, necessary and sufficient to ensure that "soft-wired" one-shot types cannot be boosted. This condition yields simple proofs that one-shot deterministic types cannot be boosted, that k -set agreement types cannot be boosted, and that M -ported objects in an N -process system cannot be boosted if $M < N$. For types that can be boosted, we derive bounds on the amount by which the consensus number can be increased. For finite types, these properties and bounds are computable.

1 Introduction

A *data object* provides a set of *operations* that manipulate its *state*. For example, a queue object's might provide 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. We think of a concurrent object as providing a finite set of *ports* through which the processes can access the object. Concurrent objects are required to be *linearizable* [8]: operations may overlap in time, but they 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, cache misses, page faults, communication delays, and failures. If one process fails or is interrupted while an operation is in progress, other processes should be able to access that object [?, ?].

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 variables, but it is possible using atomic compare-and-swap registers [6].

In the *consensus* problem [5], 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. One way to measure a type's power is by its *consensus number* [6], the maximum number of processes that can achieve consensus [5] using objects of this type together with read/write memory. More precisely, a set of types $T = \{T_0, \dots, T_\ell\}$ has consensus number N if one can construct an N -process consensus protocol using any number of objects of type $T_i \in T$ and any amount of read/write memory. One way in which consensus numbers measure synchronization power is that N -process consensus is *universal* for N -process concurrent objects: from a set of types T of consensus number N , one can construct a wait-free linearizable implementation of any N -process concurrent object [6].

In this paper, we address the following question: to what extent is the consensus number of a set of types $T = \{T_0, \dots, T_\ell\}$ determined by the consensus numbers of the T_i alone? The consensus number of T , written $c(T)$, is at least $\max(c(T_i))$, but can it be higher? The question, known as the *robustness* problem [9], is important because it determines the extent to which we can reason about the synchronization power of types in isolation.

More precisely, a type T can be *boosted* to consensus number M , if there exists a type B , called the *booster type*, such that $c(B) < M$, but $M \leq c(B, T)$: B and T together have consensus number at least M . A boosting is *non-trivial* if $c(T) < M$. A class \mathcal{C} of types is *robust* if no type in 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 do exist examples of objects with low consensus number that can be combined to implement objects of higher consensus number [4, 11, 12, 14, 17]. Nevertheless, the practical importance of these examples remains unclear. For example, 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 [16].

In this paper, we show that any type that can be boosted must satisfy properties unlikely to occur in any realistic computer architecture. Violations of robustness, while mathematically

interesting, thus seem quite unlikely to arise in practice.

We focus on base types from the class of *one-shot* types, encompassing objects that permit each process to invoke one operation. (All boostable types cited above are one-shot types.) We give a new and simple condition, called *solo-connectivity*, sufficient to ensure that one-shot types cannot be boosted. (The booster type need not be one-shot, and can be arbitrary.) 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.

2 Model

Our model is similar to that of Jayanti [11]. 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*). Each port of a *one-shot* object can be used only once by a unique process. A process may use more than one port of a one-shot object during an execution.

In this paper, we restrict our attention to one-shot objects with soft-wired ports. 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 [8].

The behaviour of a one-shot object can be described using notions taken from combinatorial topology (for details, see [7]). Throughout this paper, we assume there are $N = n + 1$ processes. (It is convenient to use n for topological dimension, and N for the number of processes.)

A *vertex* \vec{v} is a point in a high-dimensional Euclidean space. Vertexes $\vec{v}_0, \dots, \vec{v}_n$ are *affinely independent* if $\vec{v}_1 - \vec{v}_0, \dots, \vec{v}_n - \vec{v}_0$ are linearly independent. An *n -dimensional simplex* (or *n -simplex*) $S^n = (\vec{s}_0, \dots, \vec{s}_n)$ is the convex hull of a set of $n + 1$ affinely-independent vertexes. 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 T if the vertexes of S are a (proper) subset of the vertexes 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 $\vec{s}_0, \dots, \vec{s}_k$ such that each $(\vec{s}_i, \vec{s}_{i+1})$ is a 1-simplex of \mathcal{K} . A *component* of a complex is a maximal subcomplex of \mathcal{K} containing vertexes reachable from one another by a path. A complex is *connected* if all vertexes lie in the same component.

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

For each object state q , an $(n + 1)$ -process one-shot type T thus defines an *invocation complex* \mathcal{I}_q , a *result complex* \mathcal{R}_q , and a relation Δ_q carrying each d -simplex of \mathcal{I}_q to a set of matching d -simplexes of \mathcal{R} , for $0 \leq d \leq n$ that describes the possible responses to a given group of invocations. Each d -simplex I in \mathcal{I}_q is a possible set of invocations that can be applied to an object of type T . For each d -simplex I in \mathcal{I}_q , each d -simplex R^d in $\Delta_q(I)$ is a matching set of responses permitted by the object specification. It follows easily from the definition that, for any input simplexes $I' \subset I$, each simplex of $\Delta_q(I')$ is a face of some simplex in $\Delta_q(I)$. When convenient, we sometimes omit explicit mention of q .

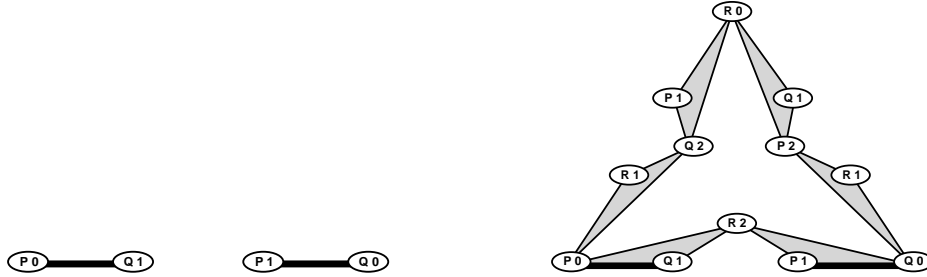


Figure 1: Fetch-and-Inc complexes

Figure 2: Two-Port (left) and three-port (right) Fetch-and-Inc complexes

2.1 Related Work

Chandra et al. [4] showed that the class of non-deterministic objects is not robust in the hard-wired model. Moran and Rappoport [14] 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 [15] also described booster types for this scenario. Schenk [17] 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. In addition, the proof uses a slightly different definition of wait-freedom, where the number of steps taken by a process must be bounded, although the bound may depend on the input to the protocol. This definition of wait-freedom is equivalent to the standard definition of wait-freedom for objects with bounded non-determinism. Lo and Hadzilacos [12] improved Schenk’s result by showing that there is an oblivious one-shot object with bounded non-determinism that can be boosted.

Ruppert [16] showed that the class of objects that are equipped only with read-modify-write operations 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 [2], though the proof has not yet appeared¹.

3 Solo-Connectivity

Definition 3.1 Let $k > 1$. A one-shot type T is *k-solo-connected* if, for any state q where at least k ports have not been used, there is a mapping $\sigma_q : \mathcal{I}_q \rightarrow \mathcal{R}$ with $\sigma_q(\vec{i}) \in \Delta_q(i)$ such that, for any invocation simplex $I = (\vec{i}_0, \dots, \vec{i}_{k-1})$, the vertices $\sigma_q(\vec{i}_0), \dots, \sigma_q(\vec{i}_{k-1})$ all lie in the same connected component of $\Delta_q(I)$.

By convention, no type is 1-solo-connected. Clearly, that any k -solo-connected type is also $(k + 1)$ -solo-connected.

To illustrate 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 every invocation at port P , $\Delta_0(i)$ consists of a single vertex labeled with P and 0, so there is only one possible choice for $\sigma(\vec{i})$. The left-hand-side of Figure 1 shows the output complex for

¹A contemporary paper making the same claim has since been withdrawn.

two ports, denoted P and Q . A two-port *fetch-and-inc* object is not 2-solo connected, because the two responses corresponding to solo executions are disconnected. The right-hand-side of Figure 1 shows the output complex for three ports, denoted P , Q , and R . A three-port *fetch-and-inc* object is 3-solo connected, because all responses corresponding to solo executions are connected in this complex.

We can now state the main result of this paper formally.

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

The proofs of the two directions of this theorem are sketched in Sections 5 and 6. We first examine other facts about solo-connected types and consider some examples.

Need one port per
proc assumption
for simulation but
converse is true
with or without
ass'n.

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

Proof: Let T be a k -solo-connected type. If T could be used with registers to solve k -consensus, then registers would boost T to solve k -consensus (since registers have consensus number one [13]), contradicting Theorem 3.2. ■

The converse of this corollary is true for deterministic one-shot types. It can be proved using a reduction to team-restricted consensus [16]. See appendix for details.

Theorem 3.4 *Any deterministic one-shot type T that is not k -solo-connected has consensus number at least k . Moreover, in the consensus protocol, no process accesses the same T object more than once.*

For a finite type T , it is computable whether T is k -solo-connected, although this question is undecidable in general [10].

Any M -ported object with $M < N$ can be treated as an N -ported object where the additional $N - M$ ports accept only the trivial operation with no arguments and no result, causing no state change. Any such type is solo-connected via the result vertexes from 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, appears in Chandra et al. [4].

Theorem 3.5 *Any deterministic one-shot type T cannot be boosted beyond its own consensus number.*

Proof: Suppose T has N ports and consensus number c . A standard bivalence argument shows that $c \leq N$. If $c = N$, T cannot be boosted to solve consensus among more than c processes, 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 solve $(c + 1)$ -consensus. ■

This result is similar to the claim by Borowsky, Gafni and Afek [?] that the consensus hierarchy is robust for all deterministic types. Their claim applies to any deterministic type T (not just one-shot ones), but covers only deterministic boosters (whereas Theorem 3.5 also rules out the existence of non-deterministic boosters). Theorem 3.5 does not contradict the construction given by Moran and Rappoport [14], in which a 3-consensus object is boosted to N -consensus, because the Moran and Rappoport construction relies in an essential way on the assumption that objects are hard-wired.

The *equality negation* object of Lo and Hadzilacos [12] 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 a single operation, $negate(v)$, taking as argument a value in $\{0, 1, 2\}$ and returning a binary value. Two $negate$ invocations with distinct arguments must return the same value, while two invocations with identical arguments must return distinct values. It is not hard to see that the resulting output complex is not 2-solo-connected. Let \vec{p}_i and \vec{q}_j the vertexes $\langle P, i \rangle$ and $\langle Q, j \rangle$, and I_{ij} the input 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 [17] is not solo-connected. Both equality negation and weak agreement objects can be boosted.

Given a non-deterministic type T that is not solo-connected, we can still use the results of this thesis to bound how much its consensus number can be boosted by any type. 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' .*

For finite types, this bound is computable.

4 Agreement Protocols

Borowsky et al. [3] 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 Borowsky et al. 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.

To handle the simulation of object types other than registers, we introduce a new consensus protocol, called *live agreement*. The algorithm for c processes uses only registers and $(c - 1)$ -consensus objects. It is not wait-free, but any outputs produced are a correct solution to consensus. Here we describe a simplified algorithm for the case $c = 2$ that uses only registers. (The general algorithm is given in the appendix.) 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.

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. If P and Q are concurrently running safe and live agreement protocols, then one agreement or the other will eventually succeed.

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 vertexes 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 simplices may be added to \mathcal{K} a finite number of times. (More precisely, the representation of \mathcal{K} stored in read/write memory is continually updated by concurrent threads, and rescanned by the processes.) As long as the input vertexes lie in distinct components of \mathcal{K} , near agreement behaves like live agreement: the protocol may fail to converge as long as both processes take the same number of steps, but will converge if one process takes fewer steps than the others. If, however, the input vertexes 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 agreement in more detail. (For the general case, see the appendix.) 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. For each possible complex \mathcal{K} , and each \vec{p} and \vec{q} in the same component, there is a canonical path between them in \mathcal{K} (say, the lexicographically first shortest path). As soon as a process observes that the two preferences lie in the same component, it begins successively adjusting its preference to move it one step along the canonical path towards the preference of the other process until it observes that the two preferences are adjacent. This procedure will terminate, since the path remains fixed as soon as all of the (finitely many) updates to the complex have happened.

5 Simulation

We now describe the simulation techniques used to prove Theorem 3.2. By hypothesis, there exists a wait-free N -process consensus protocol using the N -ported one-shot solo-connected type T and an arbitrary booster type B . Assuming that B has consensus number $b \leq N$, we will show (in the full paper) how $b + 1$ processes can simulate this consensus protocol using only objects of type B and read/write registers. The simulation yields a $(b + 1)$ -process consensus protocol, implying that B itself has consensus number at least $b + 1$. An inductive argument shows that B alone has consensus number N , and therefore does not boost T .

A key technical difference between our simulation technique and the BG simulation technique 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 B may not provide operations to inspect or copy the object's state without changing it.

5.1 Two Processes, N Threads

We first consider the case where there are more than two threads. A thread encounters three types of objects: read/write variables, objects of booster type B , and objects of one-shot solo-connected type T . Read/write variables are simulated using safe agreement exactly as in BG simulation. a process that fails in the middle of a read or write operation can block at most one thread.

5.1.1 Booster Objects

When the simulation makes a 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 reaches level 2, but Q pauses at level 1, blocking agreement. As soon P turns its attention to thread θ' , Q backs off to 0, leaving θ blocked until P executes the booster operation. If P then fails in its unsafe region at θ' , both θ and θ' will be blocked by a single process failure.

If P stops simulating a thread during a booster entry protocol, 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 input register, and blocking until the response appears in the output 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. At any time, the object's current set of invocations defines a *current input simplex* I , and a *current output complex* $\Delta(I)$. Note that both the current input simplex and current output complex change dynamically as new invocations arrive. However, the output complex is changed at most N times. When a pending invocation appears, the processes conduct a near agreement protocol on the current output complex. For example, if P observes invocation \vec{i} , it takes $\sigma(\vec{i})$ as input to the near agreement, and writes the output to a shared array $start[P]$. (The function σ is defined in Definition ??.) Process P then stores in $prefer[1][P]$ 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 $prefer[0][Q]$ have a non-empty intersection.

Process P then undertakes a sequence of live agreements. For each round i , $1 \leq i \leq N$, 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 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 a non-empty intersection, at least one thread will be unblocked if the initial near agreement succeeds. If the live agreement succeeds, P fills in the remaining responses, and waits for more invocations to appear.

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 labor 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.

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 threads 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 it observes that a thread has finished, it 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.

First, suppose both P and Q take an infinite number of steps. Because each process takes an unbounded number of steps, 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 pending T operations. If the operations are pending 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 output complex, and the live agreement protocol will terminate after a finite number of steps.

Second, suppose P alone takes an infinite number of steps. 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 have finished the protocol.

5.2 Two Processes, Two Threads

If there are two processes and two threads, then each process directly executes a thread. There is no need for a booster entry protocol, and the two-process T simulation terminates for the reasons outlined above.

5.3 M Processes, k Threads

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

Let C be a $(c - 1)$ -consensus object constructed from B objects and registers. For accesses to read/write variables, 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, that 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 c 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.

The argument that the simulation produces a correct response is identical to the two-process case. We now argue that the simulation terminates by considering two cases.

First, if some process fails, no thread can get stuck at a T object, since the live and near agreement protocols always terminate whenever a process fails. Furthermore, a process death can kill 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 either. So some thread will always continue to make progress, so all processes will eventually complete the simulation.

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

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 solve k -consensus. In this section we show the converse holds for $k > 2$. Due to space restrictions, the formal definition of B and detailed proofs appear in the appendix.

The proof technique used here was originated by Lo [?], and has been used to prove that two one-shot types could be boosted [17, 12]. However, those proofs were intimately tied to properties of the individual types they studied. Here, the technique is generalized to handle any boostable type.

6.1 Definition of Type B

Suppose T is not k -solo-connected. Let q be a state (with at least k unused ports) of T for which the conditions of Definition 3.1 are not satisfied.

We design a non-deterministic type B that will solve k -consensus, provided the processes can “convince” the object that they can implement a T object, initialized to the state q . The B object is equipped with two operations, *propose* and *extract*. The *propose*(v, p) operation takes a natural number v (which can be interpreted as an input for the consensus problem) and a port name 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 further operations.

If the processes can cooperatively compute consistent responses for the challenge invocations, they can use these as arguments to the *extract* operation, and the B object will reveal a solution to the consensus problem. 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 has become 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.

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

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 solve k -consensus.*

References

- [1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.
- [2] 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, August 1994.
- [3] E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. Technical Report TM-573, MIT LCS, November 1997.
- [4] 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.
- [5] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

- [6] M.P. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages And Systems*, 13(1):123–149, January 1991.
- [7] M.P. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 47(1), 2000.
- [8] 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.
- [9] P. Jayanti. On the robustness of Herlihy’s hierarchy. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 145–158, 1993.
- [10] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the Workshop on Distributed Algorithms and Graphs*, pages 69–84, 1992.
- [11] Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, July 1997.
- [12] W.-K. Lo and V. Hadzilacos. All of us are smarter than any of us: more on the robustness of the consensus hierarchy. In *Proceedings of the 1997 ACM Symposium on Theory of Computing*, pages 579–588, 1997.
- [13] M.C. Loui and H.H. Abu-Amara. *Memory Requirements For Agreement Among Unreliable Asynchronous Processes*, volume 4, pages 163–183. JAI Press, 1987.
- [14] 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.
- [15] 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.
- [16] E. Ruppert. Determining consensus numbers. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 93–99, August 1997. To appear in *SIAM Journal on Computing*.
- [17] Eric Schenk. The consensus hierarchy is not robust. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, page 279, 1997.

Appendix

A.1 Proof of Theorem 3.4

Theorem 3.4 The consensus number of any deterministic one-shot type that is not k -solo-connected is at least k . (Moreover, in the consensus protocol, no process accesses the same T object more than once.)

Proof: We shall show how to solve team-restricted k -consensus, a restricted version of k -consensus, where processes are divided into two non-empty teams a priori and all processes on the same team have the same input. If it is possible to solve this constrained problem, it is also possible to solve the general k -consensus problem [16].

Since T is not k -solo-connected, there is a state q with at least k unused ports and an invocation simplex $I = (\vec{v}_0, \dots, \vec{v}_{k-1})$ such that the vertices $\Delta_q(\vec{v}_0), \dots, \Delta_q(\vec{v}_{k-1})$ do not lie in the same

component of $\Delta_q(I)$. (Since T is deterministic, $\Delta_q(\vec{v}_j)$ is a single vertex and therefore the unique possible choice for $\sigma_q(\vec{v}_j)$ in the definition of k -solo-connectivity).

We give a protocol for team-restricted k -consensus for the processes $\{P_0, \dots, P_{k-1}\}$ that uses a single object of type T , initially in state q , and one register for each of the two teams. Let team A consist of the processes P_j for which $\Delta_q(\vec{v}_j)$ lies in the component Z of $\Delta_q(I)$ that contains $\Delta_q(\vec{v}_0)$. Process P_0 belongs to team A , so A is non-empty. Let team B consist of all other processes. This team is also non-empty, by the assumption that T is not k -solo-connected.

To solve consensus, each process P_j first writes its input value into its team's register. It then performs the invocation \vec{v}_j on the object of type T . 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.

The responses that all processes receive from the T object form a simplex in $\Delta_q(I)$ and are therefore in the same component of $\Delta_q(I)$. This component must contain the response vertex $\Delta_q(\vec{v}_j)$ of the first process, P_j , to access the T object. It follows that, in any execution of this protocol, all processes output the input value of the first process to access the T object. ■

A.2 Live and Near Agreement

We first give pseudocode for the two-process protocols described in Section 4. Following this, we give the general protocols for any number of processes and prove that they have the required properties.

A.2.1 The Two-Process Case

The code for two-process live agreement is given in Figure 3. To see why live agreement works, say that a process *commits* when it reads the other process's prefer register for the last time. The first process to commit decides its preferred value, which never changes thereafter. When the second process commits at some later time, it decides its own preferred value, which must be the same as the first process's decision value.

The code for two-process near agreement is given in Figure 4

A.2.2 The General Live Agreement Algorithm

A live agreement algorithm for c processes is given in Figure 5. The algorithm uses a collection of snapshot and $(c - 1)$ -consensus objects, indexed by the natural numbers. Each snapshot object $\text{row}[i]$ has c fields, one for each process. Each of the fields has two sub-fields, *prefer* and *result*. A process can only write to its own field, but can instantaneously read the entire snapshot object. Such an object can be built from ordinary registers [1, ?, ?]. Initially, every field is null.

We first argue that all non-faulty processes terminate if any process halts. Let \hat{i} be first row that the halted process does not write in. Any process that reaches iteration \hat{i} of the loop will see fewer than c processes whenever it reads $\text{row}[\hat{i}]$. Thus, no process will continue on to iteration $\hat{i} + 1$.

To prove validity (i.e. that every output value is the input value of some process), one can easily prove the invariant that any value written into row or the private variable p is the input of some process.

Finally, we show that all values produced by the live agreement protocol are identical. Consider any run where some process produces an output. Let \hat{i} be the lowest numbered loop iteration in which some process sets its output variable. Let \hat{r} be the value returned by $\text{con}[\hat{i}]$. Since at least one process does not proceed to iteration $\hat{i} + 1$, we can use the termination argument to see that every process halts either in iteration \hat{i} or $\hat{i} + 1$.

Any process that halts in iteration \hat{i} outputs \hat{r} .

```

shared value prefer[2];          // current preference
value liveAgree (value input) {
  prefer[P] = input;           // announce input
  if (prefer[Q] != null)
    while (prefer[P] != prefer[Q])
      prefer[P] = prefer[Q];   // keep flipping
  return prefer[P];
}

```

Figure 3: Two-Process Live Agreement

```

shared Vertex prefer[2];        // current preference
shared Complex K;              // shared complex
Vertex nearAgree (Vertex input) {
  prefer[P] = input;           // announce input
  if (prefer[Q] == null)
    return prefer[P];          // no contest
  Vertex p, q;
  Complex k;
  (p, q, k) = scan(prefer[P], prefer[Q], K); // scan shared complex
  while (! adjacent(p, q, k)) {
    prefer[P] = adjust(p, q, k);
    (p, q, k) = scan(prefer[P], prefer[Q], K); // rescan shared complex
  }
  return prefer[P];
}

Vertex adjust(Vertex p, Vertex q, Complex k) {
  if (p and q in same component of k)
    return p's neighbour on path(p, q, k);
  else
    // flip values
    return q;
}

```

Figure 4: Two-Process Near Agreement

```

shared snapshot row[0,1,2,...]
shared (c-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 c 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 c 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 5: c -Process Live Agreement

Consider any process P that halts in iteration $\hat{i} + 1$. Since it did not halt in iteration \hat{i} , it must have seen values written by all c processes in $\text{row}[\hat{i}]$. We consider two cases.

First suppose P 's first read of $\text{row}[\hat{i}]$ was done after every process had written to $\text{row}[\hat{i}]$. Then, at some time before this read occurred, some process Q that halted in iteration \hat{i} must have done its second read of $\text{row}[\hat{i}]$, and, prior to that, Q must have written \hat{r} into $\text{row}[\hat{i}].\text{result}$. So, process P sets p to \hat{r} before proceeding to iteration $\hat{i} + 1$.

Now suppose P 's first read sees fewer than c fields of $\text{row}[\hat{i}]$ written. Then P updates its value of p to \hat{r} before proceeding to iteration $\hat{i} + 1$.

Either way, P 's input to $\text{con}[\hat{i} + 1]$ is \hat{r} . This is true for all processes who make it to iteration $\hat{i} + 1$. Thus, all processes that halt at the end of iteration $\hat{i} + 1$ also output \hat{r} .

A.2.3 The General Near Agreement Algorithm

The protocol for c processes is given in Figure 6. It uses snapshot and $(c - 1)$ -consensus objects. The snapshot objects have the same structure as in the live agreement algorithm, above.

Notice that all of the processes (except process 1) that execute iteration i have the same preference at the start of iteration i . So, at most two different values get written into the prefer fields of $\text{row}[i]$.

We first show that all non-faulty processes produce an output if either (1) some process halts or (2) the complex eventually becomes k -dimensional.

If some process fails, the argument for termination is identical to the argument given for live agreement.

```

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

value nearAgree (Vertex input)
  // For process #1:
  p = input
  // For the other c-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 c 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 c processes have written in row[i] then
        output = new
      end if
    elsif all vertices written in row[i].pref 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 c-1 processes:
    p = result of submitting new to con2[i]

    i = i+1
  end while
  return output
end nearAgree

```

Figure 6: c -Process Near Agreement

Now, suppose the complex eventually becomes k -dimensional. To derive a contradiction, assume all processes run forever without finishing the near agreement protocol.

Consider some iteration i that is begun after the complex has acquired full dimension.

At least two non-adjacent vertices get written in $\text{row}[i]$'s prefer fields. (Otherwise the last process to read $\text{row}[i]$ would terminate). This means that process 1's value of p (\vec{v}_1) at the start of iteration i must not be adjacent to the common p value (\vec{v}_2) that the other processes have at the beginning of iteration i . Let d be the distance between \vec{v}_1 and \vec{v}_2 in the complex.

Call those processes who first read $\text{row}[i]$ before all processes have written into it *fast*, and those who do not *slow*. We consider the value of p for each process at the start of iteration $i + 1$. Let r be the result returned by $\text{con1}[i]$. (It will be either \vec{v}_1 or \vec{v}_2 .) Each fast process sets its variable new to r in iteration i . Each slow process sets its variable new either to r or to a vertex in the interior of the path between \vec{v}_1 and \vec{v}_2 . Thus the distance between the new values of any pair of processes in iteration i is at most $d - 1$. Since each process's value for p at the start of iteration $i + 1$ is chosen from among the values of the new variables in iteration i , the distance between the p values at the start of iteration $i + 1$ is at most $d - 1$.

Thus, the distance between the two p values decreases at every iteration, and they must eventually become adjacent, at which point some process will halt. This contradicts the assumption and completes the proof of termination.

The only validity constraint is that all outputs are vertices of the complex, which is obviously satisfied.

Now, we argue that all outputs are adjacent.

Let \hat{i} be the lowest numbered iteration where some process halts. Since some process does not make it to iteration $\hat{i} + 1$, all processes halt either in iteration \hat{i} or $\hat{i} + 1$. We consider two cases.

Case I: Some process halts in iteration \hat{i} because it does two reads of $\text{row}[\hat{i}]$ before some other process has written $\text{row}[\hat{i}]$. Let \hat{r} be the response returned by $\text{con1}[\hat{i}]$. Any process that halts in iteration \hat{i} outputs either \hat{r} or the other preference (which it perceives to be adjacent to \hat{r}).

We can argue exactly as in the proof for live agreement that any process that makes it to iteration $\hat{i} + 1$ must have preference \hat{r} , so all such processes will also output \hat{r} .

Case II: No process halts in \hat{i} because both of its snapshots are done before all processes have written $\text{row}[\hat{i}]$. Some process does halt in iteration \hat{i} (by the definition of \hat{i}). Any process that does halt in row \hat{i} sees both preferences (\vec{v}_1 and \vec{v}_2) used in row \hat{i} . Furthermore, it sees that they are adjacent in the complex and outputs one of them. Now consider any process that does not halt in iteration \hat{i} . Its value of p at the start of the next iteration will be either \vec{v}_1 or \vec{v}_2 . (This is true, since the only other possibility would be a point in the interior of a shortest path between \vec{v}_0 and \vec{v}_1 in the full-dimensional complex. But in the full-dimensional complex, \vec{v}_1 and \vec{v}_2 are adjacent, so such interior points don't exist.) Thus, the processes that reach iteration $\hat{i} + 1$ will either all choose \vec{v}_1 or all choose \vec{v}_2 .

Thus any output produced is adjacent (at the time it is produced) to all earlier outputs. This completes the proof of correctness for the near agreement algorithm.

A.3 Simulation Details

We must show that when P chooses an initial value for $\text{prefer}[1][P]$, there exists a linearization of the pending operations that is consistent with the responses appearing in *start*. Let I be the simplex of invocations that appear in the snapshot of the invocation registers. The responses that appear in the start array were chosen by near agreement to be adjacent vertices in $\Delta(I')$, where I' is the simplex of invocations at some earlier time. Since I' is a face of I , every simplex in $\Delta(I')$ is also in $\Delta(I)$. So any vertices that appear in the start array lie on a common maximal simplex

of $\Delta(I)$. By the definition of Δ , this means that some linearization of the operations of I includes the responses that appear in the start array.

When P must choose an extension of $prefer[i][P]$ for the initial value of $prefer[i+1][P]$ that includes all pending operations, it is clear that such an extension exists, since it is assumed that all operations are enabled at all times.

Let Π be an N -consensus protocol that uses registers and objects of type T and of type B . We show that the simulation technique yields a 2-consensus protocol Π' that uses only objects of type B and registers.

First, it is shown that both processes obtain the same response whenever they both simulate the same step of a thread. All write operations receive the response *ack*. The processes execute a safe agreement protocol to agree on the response for each simulated read operation. So processes must obtain the same response for any simulated register operation. At most one process accesses a real object of type B to simulate a thread's B operation. After doing so, the process writes the response it receives to a register, and the other process can obtain the response to the simulated operation only by reading this value. Finally, when simulating a thread's operation on an object of type T , a process either writes the response it computes into the appropriate response register of the object, or obtains its response by reading the value in the response register. It follows from the agreement property of live agreement that two processes cannot write different responses into the same response register. Thus, both processes will have the same response for the simulated operation.

Since the protocol Π is deterministic, each step of each thread is determined by the responses to the thread's previous steps and the initial state of the thread. Both processes know the initial configuration of Π . Furthermore, if both processes agree on the responses to the first i simulated steps of a thread, they will both attempt to simulate the same operation for the $(i+1)$ th step of the thread. If both processes successfully simulate the step, they will get the same response, as argued in the previous paragraph. Thus, the view of a thread's history computed by one process must be a prefix of the view computed by the other process.

It will be shown that this common view of the history of all of the threads matches a linearized execution of the protocol Π . Consider any execution E' of the simulation protocol Π' . For each of the simulated operations in this execution, we shall identify a moment in E' called the linearization point for that operation, satisfying the following two properties.

- (1) The linearization points for the simulated operations of a single thread appear in the order that the thread is programmed to perform the operations.
- (2) The simulated operations on any object could return the responses computed by the processes if they were performed in the order of their linearization points.

These two facts imply that there is a linearized execution of Π where all threads have the histories computed in E' .

We can associate with each simulated operation an interval of time starting at the moment a process first begins to simulate the operation and ending at the moment a (possibly different) process first finishes simulating the operation. For each simulated operation, the linearization point will be chosen to lie in this interval. Since no process can begin to simulate one operation of a thread before it has finished simulating the previous operations of that thread, property (1) follows easily.

The linearization points of register operations are chosen as in the BG simulation. The linearization point of a write operation is the moment when a process simulating the operation performs its write to shared memory. The linearization point of a read operation is the moment a process took

the snapshot that is agreed upon by the safe agreement module. The read operation clearly returns the value written by the write operation whose linearization point precedes the read's linearization point most closely. This proves property (2) for registers.

The linearization point of a simulated B operation is the moment when a process accesses the corresponding real B object. Property (2) follows trivially.

Finally, we describe the linearization points of simulated T operations. Consider the iterations of the loop in the `simulateT` routine in which some responses are written by either process. Let i^* be the largest loop index of any of these iterations. Suppose, without loss of generality, that process P writes the responses in iteration i^* . Let $\lambda = \text{prefer}[i^*][P]$. Linearize the operation op at the moment that a process first writes the response to op or to any operation that appears after op in λ . This defines a linearization point for every operation that receives a response.

We first show that the linearization point of op is chosen within the interval of op , thus proving that property (1) holds for the T object. By definition, the linearization point of op is no later than the time a process first writes a response to op . It must now be shown that the linearization of op occurs after its invocation is written. The linearization point of op is the moment when the response to some operation op' is first written, where op' appears no earlier in λ than op . Let i be the iteration of the `simulateT` routine in which the response to op' is written. We consider two cases.

First, suppose the response to op' is written by line (*) of the code. Then both $\text{prefer}[i][P]$ and $\text{prefer}[i][Q]$ contain op' and one of the two sequences is a prefix of λ . (This is true since, for any $i > 1$, $\text{prefer}[i][P]$ and $\text{prefer}[i][Q]$ are extensions of either the initial value of $\text{prefer}[i-1][P]$ or the initial value of $\text{prefer}[i-1][Q]$.) Thus, op appears in either $\text{prefer}[i][P]$ or $\text{prefer}[i][Q]$ when the response to op' is first written, and so op 's invocation was written at some earlier time.

Now, suppose the response to op' is written by process X in line (**) of the code. Then $\text{prefer}[i][X]$ is a prefix of λ , since it is the output of the live agreement undertaken in iteration i . So, op appears in $\text{prefer}[i][X]$ at the moment X writes the response to op' . Thus, op 's invocation was written at some earlier time.

Property (2) is clearly satisfied for T objects, since every response written is consistent with any prefix of λ that is agreed upon, and consistent with both inputs to the iteration of live agreement that never terminates, if such an iteration exists.

Finally, the correctness of the 2-consensus protocol Π' will be shown. The output value of any process is equal to the output value of some thread in the linearized execution of Π constructed above. Thus, the validity and agreement properties of Π' follow from the validity and agreement properties of Π .

A.4 Details For Section 6.1

Here, we give a formal definition of the type B .

Let q be a state (with at least k unused ports) of T for which the conditions of Definition 3.1 are not satisfied. Let U be the set of unused ports in q . For any port $p \in U$, let \mathcal{I}^p be the set of possible invocations labeled by port p . Let \mathcal{I}_q be the subcomplex of \mathcal{I} containing only of invocations that are legal when the object of type T is in state q . Let the state set be $\{\perp, \text{upset}\} \cup \mathbb{N} \cup \{(v, I, R) : v \in \mathbb{N}, I \in \mathcal{I}_q, R \subset R' \text{ for some } R' \in \Delta_q(I)\}$. These states have the following intuitive meanings. The state \perp indicates that the B object has not yet issued any challenge invocations. The state `upset` indicates that processes have accessed the B object incorrectly, and the object will give arbitrarily chosen responses to all future operations. If the state of the object is $v \in \mathbb{N}$, it indicates that the first input value to the B object was v , and that the B object has issued a set of challenge invocations that do not form a legal simplex of \mathcal{I}_q . If a B object is in a state of the form (v, I, R) ,

the components of the state should be interpreted as follows: v is the first input value provided to the B object, I is the (legal) simplex of invocations that the B object has provided as challenges so far, and R is the set of responses to the challenges that processes have provided.

We define the transition function δ for the type B mapping the current state and an operation to the set of possible pairs of new states and responses. Thus, if $(q', res) \in \delta(q, op)$, it means that the object can return the response res and change to state q' whenever it is currently in state q and the operation op is performed.

$$\begin{aligned}
\delta(\perp, propose(v, p)) &= \{(v, \vec{i}, \emptyset), \vec{i}\} : \vec{i} \in \mathcal{I}^p\} \\
\delta(\perp, extract(\vec{r})) &= \{(upset, v) : v \in \mathbb{N}\} \\
\delta(upset, propose(v, p)) &= \{(upset, \vec{i}) : \vec{i} \in \mathcal{I}^p\} \\
\delta(upset, extract(\vec{r})) &= \{(upset, v) : v \in \mathbb{N}\} \\
\delta(v, propose(v', p)) &= \{(v, \vec{i}) : \vec{i} \in \mathcal{I}^p\} \\
\delta(v, extract(\vec{r})) &= \{(v, v)\} \\
\delta((v, I, R), propose(v', p)) &= \begin{cases} \{(upset, \vec{i}) : \vec{i} \in \mathcal{I}^p\}, & \text{if } I \text{ contains an invocation la-} \\ & \text{belled by } p \text{ or } \dim(I) \geq k - 1 \\ \{(v, I \cdot \vec{i}, R), \vec{i}\} : \vec{i} \in \mathcal{I}^p, I \cdot \vec{i} \in \mathcal{I}_q\} \cup \{(v, \vec{i}) : \vec{i} \in \mathcal{I}^p, I \cdot \vec{i} \notin \mathcal{I}_q\}, & \text{otherwise} \end{cases} \\
\delta((v, I, R), extract(\vec{r})) &= \begin{cases} \{(v, I, R \cdot \vec{r}), v\}, & \text{if } R \cdot \vec{r} \subset R', \text{ for some } R' \in \Delta_q(I) \\ \{(upset, v') : v' \in \mathbb{N}\}, & \text{otherwise} \end{cases}
\end{aligned}$$

define cdot
notation

This is getting
ugly.

The object can always respond with an arbitrary element of \mathcal{I}^p when a $propose(v, p)$ operation is performed. If there has been a previous $propose$ for the same port p or if more than k $propose$ operations have already been performed, the object becomes upset. Otherwise, the object either adds the response to the invocation simplex that is stored in the state (if the challenge invocation is legal) or moves into a new state which stores only the first input value the object received (if the challenge invocation is not legal).

If an $extract$ operation upsets the object (either because no $propose$ operations have been performed or because the $extract$ operation gave an incorrect response vertex to a legal challenge), the $extract$ can return an arbitrary natural number. Otherwise, the $extract$ operation returns the first value, v , that was proposed to it.

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

Proof: Each process first proposes its input value to the B object, obtaining an invocation \vec{i} for a T object as a response. It is possible for processes to check whether the invocations form a legal simplex using only registers. If \vec{i} is legal, the process applies \vec{i} to the T object and uses the response \vec{r} to extract the solution to consensus from the B object. ■

A.5 Proofs For Section 6

Lemma A.3 *There is a permuted k -implementation of T that uses a fetch&increment object, registers and a finite number of objects of type B .*

Proof: Since we assumed $\text{cons}(B) \geq k$, it is possible to use B objects and registers to implement T in a system of k processes [6]. To obtain the required implementation, processes first access the fetch&increment object to obtain distinct numbers between 1 and k . These numbers are used as process ids to run the k -process implementation. ■

In fact, the preceding construction does not permute the responses: the ability to permute the responses will be used later in the proof.

Let Π be the permuted k -implementation of T that uses registers, fetch&increment objects and B objects. In the following, we consider only those executions of Π where at most k processes participate. Let $I_0 \in \mathcal{I}_q$ be the simplex of inputs to Π . We can define a notion of valence for configurations of the protocol Π . We include, as auxiliary information in a configuration, the simplex I_0 . This information is not known to the processes themselves, but is used in the proof of correctness to define valence. Let Z be a connected component of the complex $\Delta_q(I_0)$. We say that a configuration C of Π is Z -valent if the outputs in all executions that continue from C form a simplex in the component Z . A configuration is *univalent* if it is Z -valent for some component Z , and *multivalent* otherwise. A configuration C is *critical* if C is multivalent, but all configurations that are obtained from C by having one process take a step are univalent. We say two univalent configurations have different valences if they are not both Z -valent for some Z .

Lemma A.4 *The protocol Π has a multivalent initial configuration.*

Proof: For each invocation \vec{v} labelled by port p , let $\sigma(\vec{v})$ be the response computed by protocol Π in some solo execution by process p with input \vec{v} . Since Π is a permuted k -implementation of T , $\sigma(\vec{v}) \in \Delta_q(\vec{v})$ for each \vec{v} . By assumption, T is not a solo-connected type, so there is some input simplex I_0 containing invocation vertices \vec{v}_0 and \vec{v}_1 such that $\sigma(\vec{v}_0)$ and $\sigma(\vec{v}_1)$ lie in different components of $\Delta_q(I_0)$. Thus, the initial configuration of Π that corresponds to the input simplex I_0 is multivalent. ■

Lemma A.5 *Some execution of Π has a critical configuration.*

Proof: Consider any configuration of an execution where some process has already produced an output \vec{r} . The outputs of other processes must be adjacent to \vec{r} in $\Delta_q(I_0)$. Thus, once any process completes the protocol, the configuration of the system is univalent. By the Lemma A.4, there is a multivalent initial configuration of Π . If Π had no critical configuration, one could construct an infinite execution containing only bivalent configurations where no process ever produces an output. This contradicts the fact that termination property of Π . ■

The following lemma can be proved by a standard bivalence argument. (It is the only place in the proof where we require the assumption that $k > 2$, in order to rule out the possibility that the object accessed after the critical configuration is a fetch&increment object.)

Lemma A.6 *The next step of every process after a critical configuration of Π accesses the same object of type B .*

Proposition A.7 *There is no permuted k -implementation of T using only registers and fetch&increment objects.*

Proof: Immediate from Lemmata A.5 and A.6. ■

Definition A.8 Let $v \in \mathbb{N}$. A B object has value v if its state is either v or of the form (v, I, R) .

It is easy to check from the definition of B that, once an object has value v , it will either be upset or have value v at any later time.

The following definition will be useful in bivalence arguments. It describes a situation where a process P , running solo, cannot distinguish between two different configurations.

Definition A.9 Two configurations of Π are called *similar to process P* if the following conditions hold:

- (1) the invocation simplex I_0 of both configurations is the same,
- (2) process P is in the same state in the two configurations,
- (3) each register and each fetch&increment object has the same state in the two configurations, and
- (4) For each object of type B , either
 - (i) it is in the same state in both configurations,
 - (ii) it is upset in one of the two configurations, or
 - (iii) it has value v in both configurations.

Lemma A.10 *If two univalent configurations are similar to process P , then they have the same valency.*

Proof: Let C_1 and C_2 be the two similar configurations. We shall construct a solo execution by P from each of them that look indistinguishable to P . Since P must decide the same value in both executions, the configurations C_1 and C_2 have the same valency. It suffices to show that there is a step s that process P can take in both configurations such that the configurations C_1s and C_2s are similar to P , since one can continue extending the solo executions by P in this way until P eventually outputs a value.

If the next step of P accesses an object that is in the same state in these two configurations, any step s by P will work.

Suppose the next step by P must access a B object that is upset in one of the two configurations. Without loss of generality, suppose it is upset in configuration C_1 . Let s be some step that P can take from the configuration C_2 . Because the object accessed is upset in C_1 , the step s is a legal step from configuration C_1 as well, and the B object will still be upset in C_1s .

Finally, suppose the next step of P accesses a B object that is in state (v, I, R) in one configuration ($C\alpha_1$, without loss of generality) and state (v, I', R') or v in the other, $C\alpha_2$. If the operation is a *propose* (u, p) operation, let s be a step that returns some vertex $\vec{v} \in \mathcal{T}^p$. If the operation is an *extract*, let s be a step that returns v . The object has value v in C_1 and C_2 . Thus, in both Cs_1 and Cs_2 , the object either has value v or is upset. ■

For the remainder of this section, let C be a critical configuration of Π and let O be the object of type B that each process accesses in its first step after C . The following lemmata can each be proved using a bivalence argument.

Lemma A.11 *Let s_1 and s_2 be two steps that some process P_1 can take immediately after C . Then Cs_1 and Cs_2 have the same valency.*

Proof: It suffices to show that Cs_1 and Cs_2 are similar to any process other than P_1 . We argue this by cases.

First, suppose the operation that P_1 performs on O after C is a *propose* (v, p) . If O is in state \perp in C , then it has value v in both Cs_1 and Cs_2 . If O is upset in C , it will clearly be upset in Cs_1 and Cs_2 . Finally, if O has value v' in C then, in both Cs_1 and Cs_2 , it either has value v or is upset.

Now, suppose the operation that P_1 performs on O after C is *extract* (\vec{r}) . If the state of O in C is \perp or upset, O will be upset in Cs_1 and Cs_2 . If O has value v in C then, in both Cs_1 and Cs_2 , it either is upset or has value v .

In each of these cases, Cs_1 and Cs_2 are similar to any process other than P_1 . ■

Impt: The way I'm using the word "step", a step is determined by the process' identity and the response it receives (not state transitions of the object).
define notation $C\alpha$ as config obtained if you take sequence of steps α starting from config C . Note: this config is uniquely defined for the objects we're interested in.

Lemma A.12 *If any process takes a single step from C , the object O does not become upset.*

Proof: To derive a contradiction, suppose there is some step s_1 such that O is upset in Cs_1 . Since C is multivalent, there must be some step s_2 such that Cs_1 and Cs_2 have different valences. By Lemma A.11 the steps s_1 and s_2 are performed by different processes. However, s_2 is a legal step when the system is in configuration Cs_1 , since O is upset. So, Cs_1s_2 and Cs_2 are similar to the process that performs s_2 , contradicting Lemma A.10. ■

Lemma A.13 *The first step of each process after C is a propose operation on O .*

Proof: To derive a contradiction, suppose some process P_1 does an *extract* operation in its first step s_1 after C . Let v be the value returned in step s_1 . Since C is multivalent, there is a step s_2 such that Cs_1 and Cs_2 have different valences. By Lemma A.11, s_2 is a step by some other process P_2 .

By Lemma A.12, O is not upset in Cs_1 , so it must have value v in C . In Cs_2 , O is either upset or has value v , so the step s_1 (an *extract* with response v) is legal when the protocol is in configuration Cs_2 . In both Cs_1 and Cs_2s_1 , O either has value v or is upset. This means that Cs_1 and Cs_2s_1 are similar to P_1 , contradicting Lemma A.10. ■

Lemma A.14 *The object O has state \perp in C .*

Proof: It follows from Lemma A.12 and the fact that an object can never move from the upset state to another state that O cannot be in the upset state in C .

Suppose O has value v in C . Let s_1 and s_2 be two steps, by distinct processes P_1 and P_2 , respectively, such that Cs_1 and Cs_2 have different valences. Process P_1 's first step is a *propose*(v, p) operation, by Lemma A.13. Thus, s_1 must be a legal step when the protocol is in configuration Cs_2 , since B can always return an arbitrary invocation labelled by port p to the *propose* operation. In both Cs_1 and Cs_2s_1 , O either has value v or is upset. So, Cs_1 and Cs_2s_1 are similar to P_1 , contradicting Lemma A.10.

Thus, O must be in state \perp in C . ■

Here, we define a protocol Π' in detail and prove that it is a permuted k -implementation of T . It is a protocol for N processes. The protocol Π' uses the same set of B objects as Π , except for object O . Each is initialized to the state it has in configuration C . Protocol Π' also uses the set of registers and fetch&increment objects used by Π , initialized with the values they have in configuration C . The protocol Π' also uses one new fetch&increment object, denoted F , initially holding the value 1.

We describe the actions of a process with input \vec{r} in protocol Π' . First, the process accesses the fetch&increment object F . Let j be the response it receives. Since we are only concerned with those executions of Π' where at most k processes participate, we have $1 \leq j \leq k$. The process then runs the programme of process j of Π , starting from C . However, in the first step after C , when the process is supposed to access O , it does no shared-memory accesses and instead pretends that the response it received from O is \vec{r} . The process continues running Π until it is about to access the object O again. It will be proved, below, that this second access does to O will eventually occur and that it will be of the form *extract*(\vec{r}). Instead of performing this operation, the process outputs \vec{r} and halts.

We can associate to any run of Π' a *corresponding* run of Π starting from C , identical except that processes really do perform the operations on O (instead of just pretending to do the propose and stopping short of the second access).

We now show that Π' is a correct permuted k -implementation of T .

Lemma A.15 *In any run of Π' (with at most k participating processes), every non-faulty process eventually produces an output.*

Proof: Let P_0 be one of the non-faulty participating processes in the run of Π' . We must show that in P_0 's simulation of Π , it eventually reaches a point where it is about to access O again, and that access is of the form $extract(\vec{r})$.

To derive a contradiction, suppose that in some execution of Π' , P_0 finishes its simulation of Π without ever reaching a point where it is poised to access O again. Let α be the corresponding execution of Π . Let P_1 be the process that takes the first step after C in α . Let P_2 be some other process whose first step, $propose(v, p)$, leads to a configuration with a different valency. (Such a process exists, by Lemma A.11.) If α contains a step by process P_2 , let α' be the execution of Π obtained from α by moving the first step of P_2 to occur just after C . Otherwise let α' be the run obtained by inserting one step of process P_2 just after C . To ensure that α' is still a legal run of Π , the response returned to all $extract$ operations should be changed to v . [[[Should more details be included to explain why α' is still a legal run of Π ?]]] However, α and α' are indistinguishable to P_0 , contradicting the fact that they contain configurations with different valences. Thus, every process eventually accesses the object O a second time.

Now suppose that, in some execution of Π' , some process runs its simulation of Π and reaches a point where it is poised to perform a second $propose$ operation on O . Let α be the corresponding execution of Π . Let s_1 be the first step after C in α . Let P_1 be the process that performs s_1 . Create a new execution α' of Π by inserting, after s_1 , one step by each process that does not take any steps after C in α . The execution α' is still legal, since each of the inserted steps is a $propose$ operation, by Lemma A.13. Let s_2 be the first step that some process (other than P_1) takes after C such that Cs_1 and Cs_2 have different valences. (Note that the step s_2 can occur when the protocol is in configuration C since s_2 is a $propose$ operation.) Let α'' be the execution of Π obtained from α' by swapping s_1 with s_2 and changing the responses to all $extract$ operations to the value proposed by s_2 . This will still be a legal execution of Π since s_1 and s_2 are $propose$ operations. The configurations $C\alpha'$ and $C\alpha''$ have different valences. However, the object O is upset in both, since at least $k+1$ $propose$ operations have occurred. Furthermore, the internal state of P_0 and the state of all objects except O are identical in the two configurations, so $C\alpha'$ and $C\alpha''$ are similar to P_0 , contradicting Lemma A.10. Thus, the second operation performed on O by any process after C must be an $extract$ operation. ■

Lemma A.16 *Consider any execution of Π' with inputs forming the simplex I_0 . In the corresponding run α of Π , the set of arguments to the $extract$ operations performed by all of the processes are distinct vertices of a face of a simplex in $\Delta_q(I_0)$.*

Proof: Suppose the claim is false to derive a contradiction. Let P_0 be the process that does the last $extract$ operation in α . The object O will be upset after this operation has been performed.

Let P_1 be the first process to take a step s_1 after C in α . By Lemma A.11, there is some other process P_2 whose first step s_2 , leads to a configuration with a different valency. Construct a new run α' of Π , by inserting a step of P_2 immediately after C (or moving the step back so that it occurs immediately after C if P_2 takes a step in α'), and changing the responses to all $extract$ operations that are performed by processes other than P_0 to the value proposed by P_2 in step s_2 . The execution α' is still a legal run of Π : all $extract$ operations applied to O return the value first proposed to it, except for the $extract$ performed by P_0 , but this $extract$ can legally return any value, since it move the object into the upset state. The configurations $C\alpha$ and $C\alpha'$ are similar to P_0 , but have different valences, contradicting Lemma A.10. ■

It follows from Lemma A.16 that the protocol Π' is a correct permuted k -implementation of T .