The Impossibility of Boosting Distributed Service Resilience [*]

Paul Attie[1] Rachid Guerraoui[2] Petr Kuznetsov[3] Nancy Lynch[4] Sergio Rajsbaum[5]

(1) Department of Computer Science and Center for Advanced Mathematical Studies, American University of Beirut
(2) Distributed Programming Laboratory, EPFL
(3) Technische Universität Berlin/Deutsche Telekom Laboratories
(4) MIT Computer Science and Artificial Intelligence Laboratory
(5) Instituto de Matemáticas, Universidad Nacional Autónoma de México (UNAM)

March 17, 2010

Abstract

We study f-resilient services, which are guaranteed to operate as long as no more than f of the associated processes fail. We prove three theorems asserting the impossibility of boosting the resilience of such services. Our first theorem allows any connection pattern between processes and services but assumes these services to be *atomic (linearizable) objects*. This theorem says that no distributed system in which processes coordinate using f-resilient atomic objects and reliable registers can solve the consensus problem in the presence of $f+1$ undetectable process stopping failures. In contrast, we show that it is possible to boost the resilience of some systems solving problems easier than consensus: for example, the 2-set consensus problem is solvable for $2n$ processes and $2n-1$ failures (i.e., wait-free) using n-process consensus services resilient to $n-1$ failures (wait-free). Our proof is short and self-contained.

We then introduce the larger class of *failure-oblivious* services. These are services that cannot use information about failures, although they may behave more flexibly than atomic objects. An example of such a service is totally ordered broadcast. Our second theorem generalizes the first theorem and its proof to failure-oblivious services.

Our third theorem allows the system to contain *failure-aware* services, such as failure detectors, in addition to failure-oblivious services. This theorem requires that each failure-aware service be connected to all processes; thus, $f+1$ process failures overall can disable all the failure-aware services. In contrast, it is possible to boost the resilience of a system solving consensus using failure-aware services if arbitrary connection patterns between processes and services are allowed: consensus is solvable for any number of failures using only 1-resilient 2-process perfect failure detectors.

As far as we know, this is the first time a unified framework has been used to describe both atomic and non-atomic objects, and the first time boosting analysis has been performed for services more general than atomic objects.

[*]The first author was supported by the National Science Foundation under Grant No. 0204432. The fourth author was supported by the National Science Foundation under Grants NSF CNS-0121277 and NSF CCF-0726514 and by the Air Force Office of Scientific Research under Contracts USAF, AFRL Award No. FA9550-04-1-0121, FA9550-08-1-0159. The basic results appeared initially in a 2002 technical report [2]. An extended abstract [1] containing some of the results of this paper was presented at the 25'th International Conference on Distributed Computing Systems, June 2005, Columbus, Ohio. Part of this work was done while the third author was at the School of Computer and Communication Sciences, EPFL, and Max Planck Institute for Software Systems.

Keywords: distributed services, resilience, boosting, consensus, atomic objects, failure detectors, I/O automata.

1 Introduction

We consider distributed systems consisting of asynchronously operating processes that coordinate using shared services and reliable multi-writer multi-reader registers. A *service* is a distributed computing mechanism that interacts with distributed processes, accepting invocations, performing internal computation steps, and delivering responses. Examples of services include:

- Shared atomic (linearizable) objects, defined by sequential type specifications [11,12], for example, atomic read-modify-write, queue, counter, test&set, compare&swap and consensus objects.
- Concurrently-accessible data structures such as balanced trees.
- Broadcast services such as totally ordered broadcast and atomic broadcast [10].
- Failure detectors, which provide processes with information about the failure of other processes [5]. [1]

Thus, our notion of a service is quite general. We define three successively more general classes of service—atomic objects, failure-oblivious services, and general (possibly failure-aware) services—in Sections 2.1, 5, and 6. We define our services to tolerate a certain number f of failures: a service is f-resilient if it is guaranteed to operate as long as no more than f of the processes connected to the service fail.

This paper considers the question of what level of resilience can be achieved by distributed systems containing certain kinds of services. In particular, we prove results saying that the resilience of a system cannot be "boosted" above that of its individual services. More specifically, we prove three theorems saying that no distributed system in which processes coordinate using reliable registers and f-resilient services can solve the *consensus problem* in the presence of $f+1$ process-stopping failures.

We focus on the consensus problem because it is fundamental to the study of resilience in distributed systems. For example, consensus has been shown to be universal [11], in the sense that an atomic object of any sequential type can be implemented in a *wait-free* manner (i.e., tolerating any number of failures), using (an infinite number of) wait-free consensus objects. In fact, the choice of consensus is crucial because our results do not apply to some problems that are weaker than consensus, such as k-set-consensus.

Our contribution. Our first main theorem, Theorem 2, assumes that the given services are *atomic objects* and allows any connection pattern between processes and services. This theorem is a strict generalization of the classical impossibility result of Fischer et al. [8] for fault-tolerant consensus. The proof is short and self-contained. It uses a bivalence argument inspired by (though somewhat more elaborate than) the one in [8]. The proof involves showing that decisions can be made in a particular way, described by a *hook* pattern of executions.

In contrast to the impossibility of boosting for consensus, we show that it *is* possible to boost the resilience of systems solving problems easier than consensus. In particular, we show that the 2-set

[1] Our notion of service allows us to model some of the failure detectors defined by Chandra et al. [4]. See Section 6.2 for examples.

consensus problem is solvable for $2n$ processes and $2n-1$ failures (i.e., wait-free) using n-process consensus services resilient to $n-1$ failures (i.e., wait-free).

Theorem 2 and its proof assume that the given services are atomic objects; however, they extend to a larger class of *failure-oblivious* services. A failure-oblivious service generalizes an atomic object by allowing an invocation to trigger any number of responses, to any number of processes. The service may also perform spontaneous steps, not triggered by an invocation. The key constraint is that no step may depend on explicit knowledge of the occurrence of failure events. We define the class of failure-oblivious services, give an example (totally-ordered broadcast), and prove a second theorem, Theorem 9, which extends Theorem 2 to failure-oblivious services.

Our third theorem, Theorem 10, addresses the case where the system may contain *general* (possibly *failure-aware*) services, such as failure detectors or atomic broadcast services, in addition to failure-oblivious services and reliable registers. This result also says that boosting is impossible. However, it requires the additional assumption that each general service is connected to *all* processes; thus, $f+1$ process failures overall can disable all the general services. The proof is a variant of our second proof, again using a "hook" construction. We also show that the stronger connectivity assumption is necessary, by demonstrating that it *is* possible to boost the resilience of a system solving consensus if arbitrary connection patterns between processes and general services are allowed: specifically, consensus is solvable for any number of failures using only 1-resilient 2-process perfect failure detectors.

In addition to the three main theorems, our paper presents (as far as we know) the first unified framework for expressing both atomic and non-atomic objects. Our models for failure-oblivious services and general services are new. Moreover, this is the first time boosting analysis has been performed for services more general than atomic objects.

Related work. Our Theorem 2 asserting the impossibility of boosting the resilience of atomic objects appeared initially in a 2002 technical report [2]. Subsequently, it was observed in [9,13] that a variant of Theorem 2 can be derived by combining several earlier theorems, including Herlihy's result on universality of consensus [11], observations on implementability of consensus by Jayanti and Toueg [14], and results of Chandra et al. on f-resilience vs. wait-freedom [3]. However, the models differ in some technical aspects. Some of the differences between the models are:

1. Jayanti and Toueg [14] and Herlihy [11] assume that an access to a wait-free object takes place instantaneously, while Chandra et al. [3] and our paper assume that an access to every service (even wait-free) incurs a delay. Also, Chandra et al. [3] and our paper allow a process to access multiple services concurrently.

2. In our definition of an f-resilient atomic object, a connected process P_i that does not apply an invocation is considered alive until a $fail_i$ action arrives. The corresponding definition of a weakly f-resilient object in Chandra et al. [3] counts such a process as faulty.

3. We assume that an implementation can use only finitely many services. Chandra et al. [3] does not make this assumtion.

We suspect, however, that it should not be difficult to address the points above and derive our result on atomic objects from [3, 11, 14]. Nevertheless, some of the proofs upon which such an indirect derivation (sketched in [9]) may rest are more complex than our direct proof. Also, unlike our proof, the indirect arguments do not extend to prove the impossibility of boosting for failure-oblivious and failure-aware services.

Organization. The rest of the paper is organized as follows. Section 2.1 presents definitions for the underlying model of distributed computation and for atomic objects. Section 2.2 presents our model for a system whose services are atomic objects. Section 3 presents the first impossibility result. Section 4 shows that boosting is possible for set consensus. Section 5 defines failure-oblivious services, gives an example, and proves our second impossibility result, which extends the first impossibility result to systems with failure-oblivious services. Section 6 defines general services, gives examples, and presents our third impossibility result. Section 7 concludes the paper. Appendix A provides the complete proofs for the main lemmas in the second impossibility result, for failure-oblivious services.

2 Mathematical Preliminaries

2.1 Model of distributed computation

In this section, we review definitions for basic notions used in this paper: I/O automata and sequential types. We also give new definitions for resilient atomic objects, in terms of I/O automata.

2.1.1 I/O Automata

We use the I/O automaton model of Lynch and Tuttle [16,18], as presented in [17, Chapter 8], as our underlying model for concurrent computation. An I/O automaton is a state machine where each transition is labelled with an action name, i.e., transitions are triples (s, a, s') where s, s' are states and a is an action name. A distinguished subset of the states are start states. An execution is an alternating sequence $s_0 a_1 s_1 a_2 s_2 \ldots$ of states and actions that s_0 is a start state and every triple $s_{i-1} a_i s_i$ along the execution is an actual transition. Each action of an I/O automaton is either an input action, output action, or internal action. These three sets of actions constitute the signature of the I/O automaton. In a concurrent composition of several I/O automata, all I/O automata with an action a in their signature must execute a concurrently for a to occur. Concurrency is thus modelled as the nondeterministic interleaving of action executions. In a concurrent composition, an action a can be an output action of at most one automaton, and if a is an internal action of some I/O automaton, then it cannot be an action of any other I/O automaton. I/O automata are input-enabled, that is, from every state there is at least one transition for every input action. So, an I/O automaton has no control over the occurrence of its input actions, but does have control over the occurrence of its output and internal actions, which are collectively referred to as locally-controlled actions. The locally controlled actions of an I/O automaton are partitioned into tasks. We use associated terminology from [17, Chapter 8] as needed.

We say that an I/O automaton A is *deterministic* if and only if, for each task e of A, and each state s of A, there is at most one transition (s, a, s') such that $a \in e$.

An execution α of A is *fair* iff for each task e of A: (1) if α is finite, then e is not enabled in the final state of α, and (2) if α is infinite, then α contains either infinitely many actions of e, or infinitely many occurrences of states in which e is not enabled. A *trace* of A is a sequence of external actions of A obtained by removing the states and internal actions from an execution of A. A trace of a fair execution is called a *fair trace.*

If α and α' are execution fragments of A, with α finite, such that α' starts in the last state of α, then the concatenation $\alpha \cdot \alpha'$ is defined, and is called an *extension* of α.

An I/O automaton A *implements* an I/O automaton B iff all of the following hold:

1. A and B have the same input actions and the same output actions.

2. Any (finite or infinite) trace of A is also a trace of B.
3. Any fair trace of A is also a fair trace of B.

2.1.2 Sequential types

We define the notion of a "sequential type," in order to describe allowable sequential behavior of atomic objects. The definition used here generalizes the one in [17, Chapter 9]: here, we allow nondeterminism in the choice of the initial state and the next state. Namely, sequential type $\mathcal{T} = \langle V, V_0, \mathit{invs}, \mathit{resps}, \delta \rangle$ consists of:

- V, a nonempty set of *values*,
- $V_0 \subseteq V$, a nonempty set of *initial values*,
- *invs*, a set of *invocations*,
- *resps*, a set of *responses*, and
- δ, a binary relation from $\mathit{invs} \times V$ to $\mathit{resps} \times V$ that is *total*, in the sense that, for every $(a, v) \in \mathit{invs} \times V$, there is at least one $(b, v') \in \mathit{resps} \times V$ such that $((a, v), (b, v')) \in \delta$.[2] Relation δ specifies, for each invocation and each value of the type, a response and a new value.

We sometimes use "dot" notation, writing $\mathcal{T}.V, \mathcal{T}.V_0, \mathcal{T}.\mathit{invs}$, etc. for the components of $\mathcal{T}$.

We say that $\mathcal{T}$ is *deterministic* if V_0 is a singleton set $\{v_0\}$, and δ is a mapping, that is, for every $(a, v) \in \mathit{invs} \times V$, there is *exactly one* $(b, v') \in \mathit{resps} \times V$ such that $((a, v), (b, v')) \in \delta$.

We allow nondeterminism in our definition of a sequential type in order to make our notion of "service" as general as possible. In particular, the problem of k-set-consensus can be specified using a nondeterministic sequential type, but not a deterministic sequential type.

Example. Read/write *sequential type:* Here, V is a set of "values", $V_0 = \{v_0\}$, where v_0 is a distinguished element of V, $\mathit{invs} = \{\mathit{read}\} \cup \{\mathit{write}(v) : v \in V\}$, $\mathit{resps} = V \cup \{\mathit{ack}\}$, and $\delta = \{((\mathit{read}, v), (v, v)) : v \in V\} \cup \{((\mathit{write}(v), v'), (\mathit{ack}, v)) : v, v' \in V\}$. This is a deterministic sequential type.

Example. Binary consensus *sequential type:* Here, $V = \{\{0\}, \{1\}, \emptyset\}$, $V_0 = \{\emptyset\}$, $\mathit{invs} = \{\mathit{init}(v)) : v \in \{0, 1\}\}$, $\mathit{resps} = \{\mathit{decide}(v) : v \in \{0, 1\}\}$, and $\delta = \{((\mathit{init}(v), \emptyset), (\mathit{decide}(v), \{v\})) : v \in \{0, 1\}\} \cup \{((\mathit{init}(v), \{v'\}), (\mathit{decide}(v'), \{v'\})) : v, v' \in \{0, 1\}\}$. Thus, the first value is remembered, and is returned by every operation. This is also a deterministic sequential type.

Example. k-set-consensus *sequential type*, $0 < k < n$: Now V is the set of subsets of $\{0, 1, \ldots, n-1\}$ having at most k elements, $V_0 = \{\emptyset\}$, $\mathit{invs} = \{\mathit{init}(v) : v \in \{0, 1, \ldots, n-1\}\}$, $\mathit{resps} = \{\mathit{decide}(v) : v \in \{0, 1, \ldots, n-1\}\}$, and $\delta = \{((\mathit{init}(v), W), (\mathit{decide}(v'), W \cup \{v\})) : |W| < k, v' \in W \cup \{v\}\} \cup \{((\mathit{init}(v), W), (\mathit{decide}(v'), W)) : |W| = k, v' \in W\}$. Thus, the first k values are remembered, and every operation returns one of these values. This is a nondeterministic sequential type.

2.1.3 Canonical f-resilient atomic objects

Next, we define f-resilient atomic objects. For our impossibility proofs, it is convenient to model such objects as automata, and to express the f-resilience condition within the automata themselves, rather than treating it as a separate constraint.

A "canonical" f-resilient atomic object is an I/O automaton that exhibits all the allowable behavior, including concurrent behavior, that is permitted for an f-resilient atomic object. Namely, we define the *canonical f-resilient atomic object of type $\mathcal{T}$ for endpoint set J and index k*, where

[2] We also write $\delta((a, v), (b, v'))$ for $((a, v), (b, v')) \in \delta$.

- $\mathcal{T}$ is a sequential type,
- J is a nonempty finite set of *endpoints* at which invocations and responses may occur, (an endpoint is just a process index, i.e., the endpoints define the processes that can invoke operations on the atomic object)
- $f \in \mathbf{N}$ is the level of resilience, and
- k is a unique index for the service.

The object is described as an I/O automaton, in precondition/effect notation, in Figure 1.

We will use the parameter J to specify which processes are connected to the object; see Section 2.2. The parameter J allows different objects to be connected to the same set or different sets of processes. A process at endpoint $i \in J$ can issue any invocation specified by the underlying sequential type $\mathcal{T}$ and can (potentially) receive any allowable response. We allow concurrent (overlapping) operations, at the same or different endpoints. The object preserves the order of concurrent invocations at the same endpoint i by keeping the invocations and responses in internal FIFO buffers, two per endpoint (one for invocations from the endpoint, the other for responses to the endpoint). The object chooses the result of an operation nondeterministically, from the set of results allowed by the transition relation $\mathcal{T}.\delta$ applied to the invocation and the current value of *val*. The object can exhibit nondeterminism due to nondeterminism of the sequential type $\mathcal{T}$, and due to interleavings of steps for different process invocations.

We model a failure at an endpoint i by an explicit input action fail_i. We use the task structure of I/O automata and the basic definition of fair executions to specify the required resilience. Namely, for every process $i \in J$, we assume the object has two tasks, which we call the i-perform task and i-output task. The i-perform task includes the $\mathit{perform}_{i,k}$ action, which carries out operations invoked at endpoint i. The i-output task includes all the $b_{i,k}$ actions giving responses at endpoint i. In addition, every i-perform or i-output task contains a special $\mathit{dummy\_perform}_{i,k}$ or $\mathit{dummy\_output}_{i,k}$ action, which is enabled when either process i has failed or strictly more than f processes in J have failed. The $\mathit{dummy\_perform}_{i,k}$ and $\mathit{dummy\_output}_{i,k}$ actions are intended to allow, but not force, the object to stop performing steps on behalf of process i after i fails or after the resilience level has been exceeded.

The definition of fairness for I/O automata says that each task must get infinitely many turns to take steps. In this context, this implies that, for every $i \in J$, the object eventually responds to an outstanding invocation at i, unless either i fails or more than f processes in J fail. If i does fail or more than f processes in J fail, the fairness definition allows the object to perform the $\mathit{dummy\_perform}_{i,k}$ action every time the i-perform task gets a turn, and to perform the $\mathit{dummy\_output}_{i,k}$ action every time the i-output task gets a turn, thereby avoiding responding to i. In particular, if more than f processes fail, the object may avoid responding to any process in J, since $\mathit{dummy\_output}_{i,k}$ is enabled for every $i \in J$. Also, if all processes connected to the object (i.e., all processes in J) fail, the object may avoid responding to any process.

Thus, the basic fairness definition for I/O automata, applied to a canonical f-resilient atomic object automaton, expresses the idea that the object is f-resilient: Once more than f of the processes connected to the object fail, the object itself may "fail" by becoming silent. However, although the object may stop responding, it never violates its safety guarantees, that is, it never returns values inconsistent with the underlying sequential type specification.

We say that a canonical f-resilient atomic object automaton A is *wait-free* (or, *reliable*), if it is $(|J|-1)$-resilient. This is equivalent to saying that (a) A is $|J|$-resilient, or (b) A is f-resilient for some $f \geq |J|-1$, or (c) A is f-resilient for every $f \geq |J|-1$.

CanonicalAtomicObject$(\mathcal{T}, J, f, k)$, where $\mathcal{T} = \langle V, V_0, invs, resps, \delta \rangle$

Signature:

Inputs:
$a_{i,k}$, $a \in invs$, $i \in J$, the invocations at endpoint i
$fail_i$, $i \in J$

Outputs:
$b_{i,k}$, $b \in resps$, $i \in J$, the responses at endpoint i

Internals:
$perform_{i,k}$, $i \in J$
$dummy\_perform_{i,k}$, $i \in J$
$dummy\_output_{i,k}$, $i \in J$

Tasks:
For every $i \in J$:
    i-perform: $\{perform_{i,k}, dummy\_perform_{i,k}\}$
    i-output: $\{b_{i,k} : b \in resps\} \cup \{dummy\_output_{i,k}\}$

State components:
$val \in V$, initially an element of V_0
$inv-buffer$, a mapping from J to finite sequences of $invs$, initially identically empty
$resp-buffer$, a mapping from J to finite sequences of $resps$, initially identically empty
$failed \subseteq J$, initially $\emptyset$

Transitions:

```
Input: $a_{i,k}$
Effect:
    add $a$ to end of $inv-buffer(i)$

Internal: $perform_{i,k}$
Precondition:
    $a = head(inv-buffer(i))$
    $\delta((a, val), (b, v))$
Effect:
    remove head of $inv-buffer(i)$
    $val \leftarrow v$
    add $b$ to end of $resp-buffer(i)$

Output: $b_{i,k}$
Precondition:
    $b = head(resp-buffer(i))$
Effect:
    remove head of $resp-buffer(i)$

Input: $fail_i$
Effect:
    $failed \leftarrow failed \cup \{i\}$

Internal: $dummy\_perform_{i,k}$
Precondition:
    $i \in failed \vee |failed| > f$
Effect:
    none

Internal: $dummy\_output_{i,k}$
Precondition:
    $i \in failed \vee |failed| > f$
Effect:
    none
```

Figure 1: A canonical atomic object.

A canonical atomic object whose sequential type is read/write is called a *canonical register*. In this paper, we will assume canonical reliable (wait-free) registers.

2.1.4 f-resilient atomic objects

An I/O automaton A is an *f-resilient atomic object* of type $\mathcal{T}$ for nonempty endpoint set J and index k, provided that it implements the canonical f-resilient atomic object S of type $\mathcal{T}$ for J and k. Here, "implements" is defined as in Section 2.1.1, in terms of the same external interface, inclusion of trace sets, and inclusion of fair trace sets. Note that clause 2 (any trace of A is also a trace of S) guarantees the atomicity of A, and clause 3 (any fair trace of A is also a fair trace of S) guarantees the f-resilience of A.

We say that A is *wait-free* (or, *reliable*), if it is $(|J|-1)$-resilient. An atomic object whose sequential type is read/write is called a *register*.

The notion of an f-resilient atomic object is useful when we talk about a distributed system implementing a specific canonical service. In this case, we can say that the system *is* the service. This enables composition of implementations: an implemented service can be seen as a canonical service in a higher-level implementation.

2.2 System Model with Atomic Objects

Our system model consists of a collection of process automata, canonical resilient atomic objects, and canonical reliable registers. For this section, we fix I, R, and K, (disjoint) finite index sets for processes, registers, and resilient atomic objects, respectively, and $\mathcal{T}$, a sequential type, representing the problem the system is intended to solve. A *distributed system* for I, R, K, and $\mathcal{T}$ is the composition of the following I/O automata (see [17, Chapter 8] for the formal definition of composition):

1. *Processes* P_i, $i \in I$,
2. *Resilient atomic objects* S_k, $k \in K$. We let $\mathcal{T}_k$ denote the sequential type, and $J_k \subseteq I$ the set of endpoints, of object S_k. We let f_k denote the level of resilience. We assume k itself is the index for the object, as in the definition of a canonical atomic object.
3. *Reliable registers* S_r, $r \in R$. We let V_r denote the value set and $(v_0)_r$ the initial value for register S_r. We let $J_r \subseteq I$ denote the set of endpoints of register S_r. We assume r is the index for the register.

We assume that processes interact only via registers and resilient atomic objects. Process P_i can invoke an operation on resilient atomic object S_k provided that $i \in J_k$. Process P_i can also invoke a read or write operation on register S_r provided that $i \in J_r$. Services (resilient atomic objects and registers) do not communicate directly with one another, but may interact indirectly via processes. In the remainder of this section, we describe the components in more detail and define terminology needed for the results and proofs.

2.2.1 Processes

We assume that process P_i, $i \in I$ has the following interface (inputs and outputs):

- Inputs a_i, $a \in \mathcal{T}.invs$, and outputs b_i, $b \in \mathcal{T}.resps$. These represent P_i's interactions with the external world.
- For every resilient atomic object S_k such that $i \in J_k$, outputs $a_{i,k}$, where $a \in invs_k$, and inputs $b_{i,k}$, where $b \in resps_k$.
- For every reliable register S_r such that $i \in J_r$, outputs $a_{i,r}$, where a is a (read or write) invocation of S_r, and inputs $b_{i,r}$, where b is a response of S_r.
- Input $fail_i$. This represents the failure of process P_i.

P_i may issue several invocations, on the same or different registers or resilient atomic objects, without waiting for responses to previous invocations. The external world at P_i may also issue several invocations to P_i without waiting for responses.

In the special case where P_i is part of an implementation of consensus or k-set-consensus, we assume, as a technicality for the proofs, that when P_i performs a $decide(v)_i$ output action, it records the decision value v in a special state component (see Section 2.2.4 for the formal definition of the consensus problem).

We assume that P_i has only a single task, which therefore consists of all the locally controlled actions of P_i. We assume that in every state, some action in that single task is enabled. This action might be a "dummy" action, as in the canonical resilient atomic objects defined in Section 2.1.3. We assume that the fail_i input action affects P_i in such a way that, from the point of the failure onward, no output actions of P_i are enabled. However, other locally controlled actions may be enabled—in fact, by the restriction just above, some such action *must* be enabled.

We do not make any other restrictions on when P_i may perform actions, for example, we do not insist that P_i take "non-dummy" steps only during certain intervals defined by external invocations and responses. Our impossibility proofs do not depend on this assumption. On the other hand, when devising implementations of atomic objects, we will typically prefer algorithms in which processes are active only during intervals between invocations and responses.

2.2.2 Resilient atomic objects and registers

We assume that resilient atomic object S_k is the canonical f_k-resilient atomic object of type $\mathcal{T}_k = \langle V_k, (V_0)_k, \mathit{invs}_k, \mathit{resps}_k, \delta_k \rangle$ for J_k and k.

We assume that register S_r is the canonical wait-free atomic read/write object with value set V_r and initial value $(v_0)_r$, for J_r and r. We write invs_r, resps_r, and δ_r for the invocations, responses, and transition relation of S_r.

2.2.3 The complete system

The complete system $\mathcal{C}$ is constructed by composing the P_i, S_r, and S_k automata in parallel and then hiding the actions used to communicate among these automata. Our parallel composition and hiding operations are the standard I/O automata operations defined in [17, Chapter 8].

When we compose the automata, invocation outputs of process P_i of the form $a_{i,c}$, $c \in K \cup R$, "match up" with corresponding invocation inputs of service S_c, and similarly for responses. Also, a fail_i input is both an input to process P_i and an input to every service S_c for which i is an endpoint, $i \in J_c$. Thus, fail_i both causes P_i to fail and allows each service S_c for which $i \in J_c$ to stop processing on behalf of P_i. It also adds to the tally of failures recorded by S_c.

We now consider tasks in the composed system. As we specified earlier, each process P_i has a single task, consisting of all the locally controlled actions of P_i. Each service S_c, $c \in K \cup R$, has two tasks for each $i \in J_c$: i-perform, consisting of $\{\mathit{perform}_{i,k}, \mathit{dummy\_perform}_{i,k}\}$, and i-output, consisting of $\{b_{i,k} : b \in \mathit{resps}_k\} \cup \{\mathit{dummy\_output}_{i,k}\}$. These tasks define a partition of the set of all actions in the system, except for the inputs of the process automata that are not outputs of any other automata, namely, the invocations by the external world and the fail_i actions. The I/O automata fairness assumption says that each of these tasks gets infinitely many turns to execute.

We say that a task e is *applicable* to a finite execution α iff some action of e is enabled in the last state of α. The following lemma says that any applicable task remains applicable until an action in that task occurs.

Lemma 1 *Let α be any finite failure-free execution of $\mathcal{C}$, e be any task of $\mathcal{C}$ applicable to α, and $\alpha \cdot \beta$ be any finite failure-free extension of α such that β includes no actions of e. Then e is applicable to $\alpha \cdot \beta$.*

Proof: Task e is either a process task or service (resilient atomic object or register) task. If e is a process task, then e is applicable to any finite execution, by our assumption that each process always has some enabled locally controlled action.

If e is a service task, say of service S_c, $c \in K \cup R$, then applicability of e to α means that service S_c has either a pending invocation in an $inv-buffer$ or a pending response in a $resp-buffer$, immediately after α. Since β does not include any actions of e, and the invocation or response remains pending as long as e is not scheduled, e is also applicable to $\alpha \cdot \beta$. $\square$

For any action a of $\mathcal{C}$ and any automaton P_i, S_k, or S_r, we say that the automaton *participates* in a if it has a in its signature. For any action a of $\mathcal{C}$, we define the *participants* of a to be the set of automata that participate in a. Note that no two distinct services (resilient atomic objects or registers) participate in the same action, and similarly no two distinct processes participate in the same action. Furthermore, for any action a (except $fail_i$ actions), the number of participants is at most two. Thus, if an action a has two participants, they must be a process and a service.

2.2.4 Solving the f-resilient consensus problem

Now we can describe what it means for a distributed system to solve the f-resilient consensus problem.

The traditional specification of f-resilient binary consensus is given in terms of a set $\{P_i, i \in I\}$ of processes, each of which starts with some value v_i in $\{0, 1\}$. Processes are subject to stopping failures, which prevent them from producing any further output.[3] As a result of engaging in a consensus algorithm, each nonfaulty process eventually "decides" on a value from $\{0, 1\}$. The behavior of processes is required to satisfy the following conditions (see, e.g., [17, Chapter 6]):

Agreement No two processes decide on different values.

Validity Any value decided on is the initial value of some process.

Termination In every fair execution in which at most f processes fail, all nonfaulty processes eventually decide.

In this paper, we specify the consensus problem differently, following a style similar to that of Herlihy [11]. Namely, we identify the f-resilient binary consensus problem for a given endpoint set I with the canonical f-resilient atomic object of type consensus, for endpoint set I. We say that a distributed system $\mathcal{C}$ *solves f-resilient consensus for I* if and only if $\mathcal{C}$ is an f-resilient atomic object (as defined in Section 2.1.4) of sequential type binary consensus,[4] for endpoint set I, that is, if $\mathcal{C}$ implements the canonical f-resilient atomic object of type consensus, for endpoint set I. This definition formulates consensus as a special case of an atomic object, and thus fits consensus within the framework of this paper. In [2], we showed that any system that satisfies our definition satisfies a slight variant of the traditional one. In this variant, inputs arrive explicitly via *init*() actions, not all nonfaulty processes need receive inputs, and only nonfaulty processes that do receive inputs are guaranteed to eventually decide. Our agreement and validity conditions are the same as above. Our modified termination condition is:

Modified Termination In every fair execution in which at most f processes fail, any nonfaulty process *that receives an input* eventually decides.

[3]Stopping failures are usually defined as disabling the process from executing at all. However, the two definitions are equivalent with respect to overall system behavior.

[4]We use consensus for binary consensus in the sequel.

Thus, although we have defined "solving consensus" formally in terms of canonical atomic objects, we will use the agreement, validity, and modified termination conditions freely in our impossibility proofs. Appendix B summarizes the proof from [2] that our "operational" definition implies this variant of the usual traditional axiomatic definition.

3 Impossibility of Boosting for Atomic Objects

Our first theorem is:

Theorem 2 *Let I be a set of endpoints, $n = |I|$, and let f be an integer such that $0 \leq f < n - 1$. There is no distributed system using only canonical f-resilient atomic objects and canonical reliable registers that solves $(f + 1)$-resilient binary consensus for I.*

To prove Theorem 2, we assume that such an implementation exists and derive a contradiction. Let $\mathcal{C}$ denote the complete system, that is, the composition of the processes P_i, $i \in I$, resilient atomic objects S_k, $k \in K$, and reliable registers S_r, $r \in R$. Here I, K, and R are sets of indices that we use to provide unique names for each process, service, and register, respectively. By assumption, $\mathcal{C}$ implements an $(f + 1)$-resilient consensus atomic object. As described in Section 2.2.4, this implies that $\mathcal{C}$ satisfies the traditional agreement and validity properties of consensus, and the new termination property.

For each component index $c \in K \cup R$ and each $i \in J_c$ (recall that J_c denotes the endpoints of c), let $inv-buffer(i)_c$ denote the invocation buffer of c, which stores invocations from P_i, and let $resp-buffer(i)_c$ denote the response buffer of c, which stores responses to P_i. Also let $buffer(i)_c$ denote the pair $\langle inv-buffer(i)_c, resp-buffer(i)_c\rangle$.

The rest of this section provides the proof of Theorem 2, as follows. Section 3.1 presents and justifies an assumption that processes and sequential types are deterministic. This assumption simplies our proofs. Section 3.2 defines a class of executions of $\mathcal{C}$, the finite failure-free input-first executions, that we will use in the proof, and establishes some preliminary valence results for this class of executions. Section 3.3 defines a graph $G(\mathcal{C})$ that gives relationships among the finite failure-free input-first executions of $\mathcal{C}$. It can be regarded as a transition graph. Section 3.4 shows that $G(\mathcal{C})$ contains a "hook," similarly to [8], with two endpoints of opposite valence (see Figure 2). Section 3.5 defines two similarity notions for system states: states that are "similar" except for one process, or except for one atomic service. It then presents two lemmas, one for each notion of similarity, which state that univalent finite failure-free input-first executions that end in similar states must have the same valence. Section 3.6 then uses these lemmas to show that $G(\mathcal{C})$ cannot contain a hook, which along with Section 3.4 provides the desired contradiction. The proof uses the failure of the distinguished process or atomic service to show that the two endpoints of the hook must have the same valence.

3.1 Determinism assumptions

To prove Theorem 2, we add determinism assumptions for processes and services. These are without loss of generality. First, we require the processes to be deterministic:

(i) Each process P_i, $i \in I$, is a deterministic automaton, as defined in Section 2.1.1.

For resilient atomic objects, we assume a slightly weaker condition:

(ii) Each resilient atomic object S_k, $k \in K$, has a deterministic sequential type; that is, the sequential type $\mathcal{T}_k$ has a unique initial value $(v_0)_k$ and the transition relation δ_k is a mapping.

Note that the sequential type for each register is also deterministic, by definition. Assumptions (i) and (ii) do not reduce the generality of our impossibility result, because any candidate system could be restricted, by removing transitions, to satisfy these assumptions. If the impossibility result holds for the restricted automaton, then it holds also for the original one.

Assumptions (i) and (ii) imply that, after a finite *failure-free* execution α, an applicable task e determines a unique transition, arising from running task e from the final state s of α. We denote this transition using the function notation $\mathit{transition}(e, \alpha)$, or alternatively, $\mathit{transition}(e, s)$, since it is uniquely determined by the final state s of α. Note that $\mathit{transition}(e, \alpha)$ is defined iff e is applicable to α iff (some action in) e is enabled in s. As a result, any *failure-free* execution can be generated by applying a sequence of applicable tasks, one after the other, to the initial state of $\mathcal{C}$. The task sequence is enough to uniquely specify the execution.

If $\mathit{transition}(e, \alpha) = (s, a, s')$, then we write $\mathit{action}(e, s)$ to denote a, and $e(s)$ to denote s'. We write $e(\alpha)$ to denote α extended by $\mathit{transition}(e, \alpha)$, that is, $e(\alpha) = \alpha \cdot (s, a, s')$.

Let s be any state of $\mathcal{C}$ arising after a finite failure-free execution α of $\mathcal{C}$, and let e be a task that is applicable to α (equivalently, enabled in s). Then we write $\mathit{participants}(e, s)$ for the set of participants of action $\mathit{action}(e, s)$. Note that, for any task e and any state s, $|\mathit{participants}(e, s)| \leq 2$. Also, if $|\mathit{participants}(e, s)| = 2$, then $\mathit{participants}(e, s)$ is of the form $\{P_i, S_c\}$, for some $i \in I$ and $c \in K \cup R$.

3.2 Initializations and valence

Our proof follows a general strategy inspired by, though somewhat more elaborate than, the one in [8] and [15]. As in [8,15], the first step of the proof is to produce a bivalent initial configuration.

We consider executions of $\mathcal{C}$ in which consensus inputs arrive from the external world at the beginning of the execution. Thus, we define an *initialization* of $\mathcal{C}$ to be a finite execution of $\mathcal{C}$ containing exactly one $\mathit{init}()_i$ action for each $i \in I$, and no other actions. An execution α of $\mathcal{C}$ is *input-first* if it has an initialization as a prefix, and contains no other $\mathit{init}()$ actions. A finite failure-free input-first execution α is defined to be 0-*valent* if (1) some failure-free extension of α contains a $\mathit{decide}(0)_i$ action, for some $i \in I$, and (2) no failure-free extension of α contains a $\mathit{decide}(1)_i$ action, for any $i \in I$. The definition of a 1-*valent* execution is symmetric. A finite failure-free input-first execution α is *univalent* if it is either 0-valent or 1-valent. A finite failure-free input-first execution α is *bivalent* if (1) some failure-free extension of α contains a $\mathit{decide}(0)_i$ action, for some i, and (2) some failure-free extension of α contains a $\mathit{decide}(1)_i$ action, for some i. These definitions, and the termination requirements for consensus, immediately imply the following result:

Lemma 3 *Every finite failure-free input-first execution of $\mathcal{C}$ is either bivalent or univalent.*

The following lemma provides the first step of the impossibility proof:

Lemma 4 *$\mathcal{C}$ has a bivalent initialization.*

Proof: Write $I = \{1, \ldots, n\}$. For each $i \in \{0, \ldots, n\}$, let α^i be an initialization of $\mathcal{C}$ in which processes $P_1, \ldots, P_i$ receive initial value 1 and processes $P_{i+1}, \ldots, P_n$ receive 0. By the validity property of $\mathcal{C}$ and Lemma 3, α^0 is 0-valent, α^n is 1-valent, and every α^j ($j \in \{0, \ldots, n\}$) is either univalent or bivalent.

Then there must be some index $i \in \{0, \ldots, n-1\}$ such that α^i is 0-valent and α^{i+1} is either 1-valent or bivalent. The only difference between the initializations in α^i and α^{i+1} is the initial value

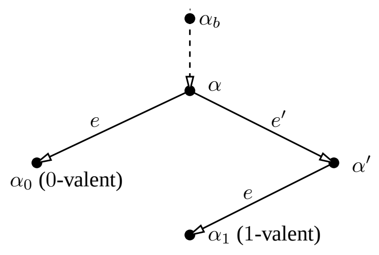

Figure 2: A hook starting in α.

of P_i. So consider a failure-free extension of α^i that is fair, except that P_i takes no steps. Since this execution looks to the rest of the system like an execution in which P_i has failed, the termination condition requires that the other processes must eventually decide, as $\mathcal{C}$ is $(f+1)$-resilient, $f \geq 0$. Since the finite execution up to the point where the processes decide is in fact failure-free, and α^i is 0-valent, the decision must be 0.

Now, an analogous failure-free extension may be constructed for α^{i+1}, also leading to a decision of 0. Since, by assumption, α^{i+1} is either 1-valent or bivalent, it must be bivalent. □

For the rest of Section 3, fix α_b to be any particular bivalent initialization of $\mathcal{C}$.

3.3 The graph $G(\mathcal{C})$

Next, we define an edge-labeled directed acyclic graph (actually, a directed tree) $G(\mathcal{C})$ to represent relationships among finite failure-free input-first executions of $\mathcal{C}$:

(1) The vertices of $G(\mathcal{C})$ are the finite failure-free input-first extensions of the bivalent initialization α_b.

(2) $G(\mathcal{C})$ contains an edge labeled with task e from α to α' provided that $\alpha' = e(\alpha)$; that is, e is applicable to α and α' is the resulting extended execution.

By assumptions (i) and (ii) of Section 3.1, any task triggers at most one transition after a failure-free execution α. Therefore, for any vertex α of $G(\mathcal{C})$ and any task e, $G(\mathcal{C})$ contains at most one edge labeled with e outgoing from α.

3.4 The existence of a hook

As in [8], we show that decisions in $\mathcal{C}$ can be made in a particular way, described by a *hook* pattern of executions. Similarly to [4], we define a *hook* to be a subgraph of $G(\mathcal{C})$ of the form depicted in Figure 2. That is, from a particular finite failure-free input-first execution α, one applicable task e leads to a 0-valent extension α_0, whereas a second applicable task e' leads to an extension α', from which the first task e leads to a 1-valent extension α_1.

Lemma 5 *$G(\mathcal{C})$ contains a hook.*

Proof: The proof is derived from the corresponding one in [8], with significant modifications reflecting the applicability of tasks in $\mathcal{C}$.

Starting from the bivalent vertex α_b of $G(\mathcal{C})$, we generate a path π in $G(\mathcal{C})$ that passes through bivalent vertices only, as follows. We consider all tasks in a round-robin fashion. Suppose we have reached a bivalent execution α so far, and task e is the next task in the round-robin list that is applicable to α. (We know such a task exists because the process tasks are always applicable.)

Lemma 1 implies that, for any finite failure-free extension α' of α such that e is not executed in the suffix of α' starting in the last state of α, e is applicable to α', and hence $e(\alpha')$ is defined. In other words, e is applicable to α' for any vertex α' of $G(\mathcal{C})$ that is reachable from α in $G(\mathcal{C})$ via a path that contains no edge labeled with e. We seek such a vertex α' such that $e(\alpha')$ is bivalent. If no such vertex α' exists, the path construction terminates. Otherwise, we proceed to $e(\alpha')$ and continue by processing the next task in the round-robin order. This construction is presented in Figure 3. Each completed iteration of the loop extends the path by at least one edge. Let π be the path generated by this construction.

```
1: $\alpha \leftarrow \alpha_b$;
2: **while** true **do**
3:    let $e$ be the next task (in round-robin order) applicable to $\alpha$;
4:    **if** $\alpha$ has a descendant $\alpha'$ in $G(\mathcal{C})$ such that the path from $\alpha$ to $\alpha'$ includes no $e$ labels
          and $e(\alpha')$ is bivalent **then**
5:       choose some such $\alpha'$;
6:       $\alpha \leftarrow e(\alpha')$
7:    **else**
8:       exit
```

Figure 3: Hook location in $G(\mathcal{C})$.

First, suppose that π is infinite. Then π corresponds to a fair failure-free input-first execution α of $\mathcal{C}$. Moreover, every finite input-first prefix of α is bivalent. Thus, no process can decide in α (for otherwise, the agreement property of $\mathcal{C}$ would be violated). This contradicts the termination requirement for consensus. So π must be finite.

Let α be the last vertex of π. By construction, α is bivalent. The fact that the path construction terminated in α means that there must be some particular task e (the next one in the round robin order that is applicable to α) satisfying the following condition: For any descendant α' of α in $G(\mathcal{C})$ such that the path from α to α' includes no e labels, $e(\alpha')$ is univalent. Without loss of generality, assume that $e(\alpha)$ is 0-valent.

Since α is bivalent, there is a descendant α' of α in which some process decides 1. Let $\sigma_0, \ldots, \sigma_m$ be the sequence of vertices of $G(\mathcal{C})$ on the path from α to α', where $\sigma_0 = \alpha$ and $\sigma_m = \alpha'$. For each j, $0 \leq j \leq m-1$, let e_j be the label of the edge on this path from σ_j to σ_{j+1}. Thus, $\sigma_{j+1} = e_j(\sigma_j)$. Note that it is possible that e occurs as a label on the path from α to α'; that is, one or more of the e_j labels may be equal to e.

We consider two cases. First, suppose that e *does not* occur on the path from α to α'. Then e is applicable to every σ_j, $0 \leq j \leq m$, by Lemma 1. By construction, $e(\sigma_0)$ is 0-valent, $e(\sigma_m)$ is 1-valent, and every $e(\sigma_j)$, $j \in \{1, \ldots, m-1\}$, is univalent. Thus, there exists an index $j \in \{0, \ldots, m-1\}$ such that $e(\sigma_j)$ is 0-valent and $e(\sigma_{j+1})$ is 1-valent. As a result, we obtain a hook (Figure 2) with e in the hook equal to e in this proof, $\alpha = \sigma_j$, $\alpha' = \sigma_{j+1}$, $\alpha_0 = e(\sigma_j)$, $\alpha_1 = e(\sigma_{j+1})$, and $e' = e_j$.

Second, suppose that e *does* occur on the path from α to α'. By our determinism assumptions (Section 3.1), the task e_0 labeling the first edge on this path is not e. Choose k to be the smallest index such that $e_k = e$; then $1 \leq k \leq m-1$. Then e is applicable to every σ_j, $0 \leq j \leq k$, again by Lemma 1. We know that $e(\sigma_0)$ is 0-valent and every $e(\sigma_j)$, $j \in \{1, \ldots, k\}$, is univalent. Furthermore, $e(\sigma_k) = \sigma_{k+1}$ must be 1-valent, because α', in which someone decides 1, is a descendant of σ_{k+1}. Thus, there exists an index $j \in \{0, \ldots, k-1\}$ such that $e(\sigma_j)$ is 0-valent and $e(\sigma_{j+1})$ is 1-valent. From this, we can construct a hook as in the first case. □

3.5 Similarity

In this section, we define some notions of similarity between system states. These will be used in obtaining a contradiction to the existence of a hook, which will yield our impossibility result.

Our similarity definitions capture what it means for system states to "look the same" to all system components except for one particular process j, or to all components except for one resilient atomic object k. The first of these notions was present implicitly in the proof of [8], but not extracted formally; the second is new here. We prove two lemmas about our similarity definitions, showing that similar states must lead to the same decision value. Our lemmas about similarity encapsulate reasoning about executions and valence, so that the main proof can focus exclusively on what happens in a few individual steps.

First, we define j-similar system states, for a process index j. Let $j \in I$ and let s_0 and s_1 be states of $\mathcal{C}$. Then s_0 and s_1 are *j-similar* if the following hold:

(1) For every $i \in I - \{j\}$, the state of P_i is the same in s_0 and s_1.

(2) For every $c \in K \cup R$:

1. The value of val_c is the same in s_0 and s_1.
2. For every $i \in J_c - \{j\}$, the value of $buffer(i)_c$ is the same in s_0 and s_1.

That is, the state of every process except for P_j is the same in s_0 and s_1, and the state of every service is also the same, except possibly for the portions of the services devoted to invocations and responses of j. The following key lemma says that, if two univalent executions end in j-similar states, for any particular j, then they must have the same valence.

Lemma 6 *Let $j \in I$. Let α_0 and α_1 be finite failure-free input-first executions, s_0 and s_1 the respective final states of α_0 and α_1. Suppose that s_0 and s_1 are j-similar. If α_0 and α_1 are univalent, then they have the same valence.*

Proof: We proceed by contradiction. Fix j, α_0, α_1, s_0, and s_1 as in the hypotheses of the lemma, and suppose (without loss of generality) that α_0 is 0-valent and α_1 is 1-valent. Let $J \subseteq I$ be any set of indices such that $j \in J$ and $|J| = f+1$ (recall that f is the resilience of the services in the system, as defined in the statement of Theorem 2). Since $f < n-1$ by assumption, we have $|J| < n$, and so $I - J$ is nonempty.[5]

Consider a fair extension of α_0, $\alpha_0 \cdot \beta$, in which the first $f+1$ actions of β are $fail_i$, $i \in J$, and no other *fail* actions occur in β. Note that, for every $i \in J$, β contains no output actions of P_i. Assume that in β, no $perform_{i,c}$ or $b_{i,c}$ ($b \in resps_c$) action for any $i \in J$, occurs at any component $c \in K \cup R$; we may assume this because, for each $i \in J$, action $fail_i$ enables a *dummy* action in every i-perform and i-output task of every service.

Since α_0 is a failure-free input-first execution, the resulting extension $\alpha_0 \cdot \beta$ is a fair input-first execution containing $f+1$ fail actions. Therefore, by the termination property for $(f+1)$-resilient

[5]The choice of J could be slightly simpler in this proof; we chose it in this way to facilitate the extension to the failure-aware case.

consensus and the fact that $I - J$ is nonempty, there is a finite prefix of $\alpha_0 \cdot \beta$, which we denote by $\alpha_0 \cdot \gamma$, that includes $\mathit{decide}(v)_l$ for some $l \notin J$ and $v \in \{0, 1\}$. Construct $\alpha_0 \cdot \gamma'$, where γ' is obtained from γ by removing the fail_i action and all subsequent internal actions of P_i, for every $i \in J$, plus all *dummy* actions. Thus, $\alpha_0 \cdot \gamma'$ is a failure-free extension of α_0 that includes $\mathit{decide}(v)_l$ (it is failure-free since all the fail_i actions have been removed). Since α_0 is 0-valent, v must be equal to 0.

Next, we claim that $\mathit{decide}(0)_l$ occurs in the suffix γ', rather than in the prefix α_0. Suppose for contradiction that the $\mathit{decide}(0)_l$ action occurs in the prefix α_0. Then by our technical assumption about processes, the decision value 0 is recorded in the state of P_l. Since s_0 and s_1 are j-similar and $l \neq j$, the same decision value 0 appears in the state s_1. But this contradicts the assumption that α_1, which ends in s_1, is 1-valent. So, it must be that the $\mathit{decide}(0)_l$ occurs in the suffix γ'.

Now we show how to append "essentially" the same γ' after α_1. The definition of j-similarity and the fact that $j \in J$ imply that:

(a) For every $i \notin J$, the state of P_i is the same in s_0 and s_1.

(b) For every $c \in K \cup R$,

1. The value of val_c is the same in s_0 and s_1 (that is, in the final states of α_0 and α_1).
2. For every $i \in J_c - J$, the value of $\mathit{buffer}(i)_c$ is the same in s_0 and s_1.

We know that, for every $i \in J$, γ' contains no locally controlled actions of P_i, and contains no $\mathit{perform}_{i,c}$ or $b_{i,c}$ actions ($b \in \mathit{resps}_c$), for any $c \in K \cup R$. Therefore:

(c) If γ' contains any locally controlled actions of a process P_i, then the state of P_i is the same in s_0 and s_1 (since $i \notin J$ in this case).

(d) For every $c \in K \cup R$,

1. The value of val_c is the same in s_0 and s_1.
2. For every $i \in J_c$, if γ' contains any $\mathit{perform}_{i,c}$ or $b_{i,c}$ ($b \in \mathit{resps}$) actions, then the value of $\mathit{buffer}(i)_c$ is the same in s_0 and s_1 (since $i \notin J$ in this case).

It follows that it is possible to append "essentially" the same γ' after α_1, resulting in a failure-free extension of α_1 that includes $\mathit{decide}(0)_l$. Since α_1 is 1-valent, this is a contradiction.[6]

□

Similarly, we define the notion of k-similar states, for a resilient atomic object k. Let $k \in K$, and let s_0 and s_1 be states of $\mathcal{C}$. Then s_0 and s_1 are *k-similar* if the following hold:

(1) For every $i \in I$, the state of P_i is the same in s_0 and s_1.

(2) For every $c \in (K - \{k\}) \cup R$, the state of S_c is the same in s_0 and s_1.

That is, the state of every process is the same in s_0 and s_1, and the state of every service except for S_k is also the same. The following lemma says that, if two univalent executions end in k-similar states, for any particular k, then they must have the same valence.

[6]More precisely, we append another execution fragment γ'' after α_1—the one that is generated by applying, after α_1, the same sequence ρ of tasks that generates γ'. We can prove, by induction on the number of tasks, that when ρ is applied after α_0 and α_1, in each pair of corresponding states: (a) for each $i \in I$ for which the unique task of P_i occurs in ρ, the states of P_i are the same, (b) for each $c \in K \cup R$, the values of val_c are the same, and (c) for each $c \in K \cup R$ and each $i \in J_c$ such that an i-perform or i-output task of S_c occurs in ρ, the values of $\mathit{buffer}(i)_c$ are the same. This correspondence is enough to imply that γ'' also includes the required $\mathit{decide}(0)_l$ action.

Lemma 7 *Let $k \in K$. Let α_0 and α_1 be finite failure-free input-first executions, s_0 and s_1 the respective final states of α_0 and α_1. Suppose that s_0 and s_1 are k-similar. If α_0 and α_1 are univalent, then they have the same valence.*

Proof: We proceed by contradiction. Fix k, α_0, α_1, s_0, and s_1 as in the hypotheses of the lemma, and suppose (without loss of generality) that α_0 is 0-valent and α_1 is 1-valent. Let $J \subseteq I$ be any set of indices such that $|J| = f+1$, and, if $|J_k| \leq f+1$, then $J_k \subseteq J$, whereas if $|J_k| > f+1$, then $J \subseteq J_k$.

Consider a fair extension of α_0, $\alpha_0 \cdot \beta$, in which the first $f+1$ actions of β are fail_i, $i \in J$, and no other *fail* actions occur in β. Note that, for every $i \in J$, β contains no output actions of P_i. Assume that in β, no $\mathit{perform}_{i,k}$ or $b_{i,k}$ action ($b \in \mathit{resps}_c$) of S_k occurs; we may assume this because the $f+1$ *fail* actions enable *dummy* actions in the i-perform and i-output tasks of S_k.

To see this enabling property, we consider the two cases in our choice of J. First, if $|J_k| \leq f+1$, then $J_k \subseteq J$; that is, the $f+1$ *fail* actions fail all the endpoints of S_k. This implies that the $i \in failed$ clauses in the preconditions for $\mathit{dummy\_perform}_{i,k}$ and $\mathit{dummy\_output}_{i,k}$ are satisfied, for every i. On the other hand, if $|J_k| > f+1$, then $J \subseteq J_k$; that is, all of the $f+1$ failed indices are endpoints of S_k. This implies that the $|\mathit{failed}| > f$ clauses in the preconditions for $\mathit{dummy\_perform}_{i,k}$ and $\mathit{dummy\_output}_{i,k}$ are satisfied, for every i.

Since α_0 is a failure-free input-first execution, the resulting extension $\alpha_0 \cdot \beta$ is a fair input-first execution containing $f+1$ fail actions. Therefore, by the termination property for $f+1$-resilient consensus and the fact that $I - J$ is nonempty, there is a finite prefix of $\alpha_0 \cdot \beta$, which we denote by $\alpha_0 \cdot \gamma$, that includes $\mathit{decide}(v)_l$ for some $l \notin J$ and $v \in \{0,1\}$. Construct $\alpha_0 \cdot \gamma'$, where γ' is obtained from γ by removing the fail_i action and subsequent internal actions of P_i, for every $i \in J$, plus all *dummy* actions. Thus, $\alpha_0 \cdot \gamma'$ is a failure-free extension of α_0 that includes $\mathit{decide}(v)_l$. Since α_0 is 0-valent, v must be equal to 0.

We know that $\mathit{decide}(0)_l$ occurs in the suffix γ', rather than in the prefix α_0, by the same argument as in the proof of Lemma 6.

Now we show how to append essentially the same γ' after α_1. The definition of k-similarity implies that:

(a) For every $i \in I$, the state of P_i is the same in s_0 and s_1.

(b) For every $c \in (K - \{k\}) \cup R$, the state of S_c is the same in s_0 and s_1.

We know that γ' contains no locally controlled actions of service S_k. Therefore:

(c) For every $c \in K \cup R$, if γ' contains any $\mathit{perform}_{i,c}$ or $b_{i,c}$ actions of S_c, then the state of S_c is the same in s_0 and s_1 (since $c \neq k$ in this case).

By properties (a) and (c), it follows that it is possible to append "essentially" the same γ' after α_1 (differing only in the state of S_k) resulting in a failure-free extension of α_1 that includes $\mathit{decide}(0)_l$. But α_1 is 1-valent — a contradiction. □

3.6 The non-existence of a hook

Using the similarity notions developed in Section 3.5, we can now obtain the contradiction that proves our impossibility result. The contradiction comes in the form of a proof that $G(\mathcal{C})$ cannot contain any hooks; this directly contradicts Lemma 5.

Lemma 8 *$G(\mathcal{C})$ contains no hooks.*

Proof: By contradiction. Assume that a hook, as depicted in Figure 2, exists. Let s, s', s_0, and s_1 be the respective final states of α, α', α_0, and α_1, and let e and e' be the two tasks involved in the hook, as shown.

Since α_0 and α_1 are 0-valent and 1-valent, respectively, by Lemmas 6 and 7, s_0 and s_1 cannot be j-similar for any $j \in I$, or k-similar for any $k \in K$. In particular, we cannot have $s_0 = s_1$. Also, note that $e'(\alpha_0)$ is 0-valent, since it is an extension of a 0-valent execution. Therefore, again, by Lemmas 6 and 7, $e'(s_0)$ and s_1 cannot be j-similar for any $j \in I$, or k-similar for any $k \in K$. In particular, we cannot have $e'(s_0) = s_1$.

We establish the contradiction using a series of claims:

Claim 1: $e \neq e'$.
Suppose for contradiction that $e = e'$. Then by determinism (Assumptions (i) and (ii) in Section 3.1), we have $\alpha_0 = \alpha'$. However, α_0 is 0-valent, whereas α' has a 1-valent failure-free extension α_1 — a contradiction.

Claim 1 and Lemma 1 imply that e' is enabled from $e(s)$.

Claim 2: $\mathit{participants}(e,s) \cap \mathit{participants}(e',s) \neq \emptyset$.
Suppose for contradiction that $\mathit{participants}(e,s) \cap \mathit{participants}(e',s) = \emptyset$. Therefore, the two tasks "commute", that is, $e'(e(s)) = e(e'(s))$. In other words, $e'(s_0) = s_1$ — a contradiction.

Since $\mathit{participants}(e,s) \cap \mathit{participants}(e',s) \neq \emptyset$, a process, resilient atomic object, or register must be in the intersection. We prove three claims showing that none of these possibilities can hold, thus obtaining the needed contradiction.

Claim 3: There does not exist $i \in I$ such that $P_i \in \mathit{participants}(e,s) \cap \mathit{participants}(e',s)$.
Suppose for contradiction that $P_i \in \mathit{participants}(e,s) \cap \mathit{participants}(e',s)$, for some particular $i \in I$. Then the two actions $\mathit{action}(e,s)$ and $\mathit{action}(e',s)$ involve only P_i and the buffers $\mathit{buffer}(i)_c$, $c \in K \cup R$. Furthermore (since the same task e is used), the action $\mathit{action}(e,s')$ also involves only P_i and the buffers $\mathit{buffer}(i)_c$, $c \in K \cup R$. But then the states s_0 and s_1 can differ only in the state of P_i and in the values of $\mathit{buffer}(i)_c$, $c \in K \cup R$. This implies that s_0 and s_1 are i-similar — a contradiction.

Claim 4: There does not exist $k \in K$ such that $S_k \in \mathit{participants}(e,s) \cap \mathit{participants}(e',s)$.
Suppose for contradiction that $S_k \in \mathit{participants}(e,s) \cap \mathit{participants}(e',s)$, for some particular $k \in K$. There are four possibilities:

1. $\mathit{participants}(e,s) = \mathit{participants}(e',s) = \{S_k\}$.
   Then e and e' must be perform tasks of S_k, and so involve only the state of S_k. But then the states s_0 and s_1 can differ only in the state of S_k. So s_0 and s_1 are k-similar — a contradiction.

2. For some $i \in I$, $\mathit{participants}(e,s) = \{S_k, P_i\}$ and $\mathit{participants}(e',s) = \{S_k\}$.
   Then the two tasks commute, that is, $e'(s_0) = s_1$ — a contradiction.

3. For some $i \in I$, $\mathit{participants}(e',s) = \{S_k, P_i\}$ and $\mathit{participants}(e,s) = \{S_k\}$.
   Again, the two tasks commute, that is, $e'(s_0) = s_1$ — a contradiction.

4. For some $i, j \in I$, $\mathit{participants}(e,s) = \{S_k, P_i\}$ and $\mathit{participants}(e',s) = \{S_k, P_j\}$.
   By Claim 3, we know that $i \neq j$. Then again, the two tasks commute, so $e'(s_0) = s_1$ — a contradiction.

Note that for cases 2 and 3 above, a situation may arise in which $\mathit{action}(e,s)$ and $\mathit{action}(e',s)$ access the same buffer. In this case, it must be that one action inserts an item and the other removes a different item. Hence the tasks commute. Note that an action that removes an item first checks that the buffer is nonempty, and so cannot be executed if the buffer is empty. Thus an empty buffer does not destroy the commutativity of the two actions.

Claim 5: There does not exist $r \in R$ such that $S_r \in \mathit{participants}(e,s) \cap \mathit{participants}(e',s)$.
Suppose for contradiction that $S_r \in \mathit{participants}(e,s) \cap \mathit{participants}(e',s)$, for some particular $r \in R$. There are four possibilities:

1. $\mathit{participants}(e,s) = \mathit{participants}(e',s) = \{S_r\}$.
   Then e and e' must be perform tasks of register S_r. Without loss of generality, suppose that $\mathit{action}(e,s)$ is $\mathit{perform}_{i,r}$ and $\mathit{action}(e',s)$ is $\mathit{perform}_{j,r}$. Since $e \neq e'$, we have $i \neq j$. We consider subcases based on whether the two operations performed are reads or writes:

   (a) $\mathit{action}(e,s)$ and $\mathit{action}(e',s)$ both perform read operations.
   Then the two tasks commute, so $e'(s_0) = s_1$ — a contradiction.

   (b) $\mathit{action}(e,s)$ performs a write operation.
   Then states s_0 and s_1 can differ only in the value of $\mathit{inv-buffer}(j)_r$ and $\mathit{resp-buffer}(j)_r$: in s_1, an invocation is missing from $\mathit{inv-buffer}(j)_r$ and an extra response appears at the end of $\mathit{resp-buffer}(j)_r$, with respect to $\mathit{inv-buffer}(j)_r$ and $\mathit{resp-buffer}(j)_r$ in s_0. So s_0 and s_1 are j-similar — a contradiction.

   (c) $\mathit{action}(e,s)$ performs a read operation and $\mathit{action}(e',s)$ performs $\mathit{write}(v)$.
   Then $e'(s_0)$ and s_1 differ only in the value of $\mathit{resp-buffer}(i)_r$ (different read responses may be appended at the end). So $e'(s_0)$ and s_1 are i-similar — a contradiction.

2. For some $i \in I$, $\mathit{participants}(e,s) = \{S_r, P_i\}$ and $\mathit{participants}(e',s) = \{S_r\}$.
   Then the two tasks commute, so $e'(s_0) = s_1$ — a contradiction.

3. For some $i \in I$, $\mathit{participants}(e',s) = \{S_r, P_i\}$ and $\mathit{participants}(e,s) = \{S_r\}$.
   Again, the two tasks commute, so $e'(s_0) = s_1$ — a contradiction.

4. For some $i, j \in I$, $\mathit{participants}(e,s) = \{S_r, P_i\}$ and $\mathit{participants}(e',s) = \{S_r, P_j\}$.
   By Claim 3, we know that $i \neq j$. Then the two tasks commute, so $e'(s_0) = s_1$ — a contradiction.

Now Claims 3, 4, and 5 together imply that $\mathit{participants}(e,s) \cap \mathit{participants}(e',s) = \emptyset$. But this directly contradicts Claim 2. □

Proof: (Of Theorem 2)
Lemma 5 contradicts Lemma 8. So we have derived a contradiction by assuming the negation of Theorem 2, and Theorem 2 is established. □

4 k-Set-Consensus

Theorem 2 says that it is impossible to solve $(f+1)$-resilient consensus for any endpoint set I, using any (finite) number of canonical f-resilient objects, of any types, and any (finite) number of reliable registers. This is so even if we allow arbitrary connection patterns between processes and services, so that no single set of $f+1$ failures can "silence" all of the services. Note that Theorem 2 concerns solutions to the *consensus* problem only; the situation is different for some other problems.

For example, consider the k-set-consensus problem [6], in which the processes must agree on at most $k \geq 1$ different values (k-set-consensus reduces to consensus when $k = 1$). Analogously to our treatment of consensus, we specify the f-reslient k-set-consensus problem as the canonical f-resilient atomic object of type k-set-consensus, for a given endpoint set I. We say that a distributed system $\mathcal{C}$ *solves f-resilient k-set-consensus for I* if and only if $\mathcal{C}$ is an f-resilient atomic object (as defined in Section 2.1.4) of type k-set-consensus, for endpoint set I, that is, if $\mathcal{C}$ implements the canonical f-resilient atomic object of type k-set-consensus, for endpoint set I.

Here we describe a simple distributed system $\mathcal{C}$ that solves f-resilient k-set-consensus for endpoint set $I = \{1, \ldots, n\}$, using only f'-resilient k'-set-consensus objects with n' endpoints apiece, for some particular choices of f, k, n, f', k', and n', with $f' < f$. Since $f' < f$, this boosts resilience, and shows that Theorem 2 cannot be extended, in general, to implementations of k-set-consensus.

Namely, we assume that $k'n = kn'$, $f = n - 1$, and $f' = n' - 1$. We divide the n endpoints in I into $g = k/k'$ disjoint groups, $I_1, I_2, \ldots, I_g$, each with exactly n' endpoints. For each group I_j, we use a single f'-resilient k'-consensus service S_j whose endpoints are exactly I_j. Since each k'-consensus service S_j is f'-resilient and $f' = n' - 1$, each S_j is in fact wait-free, and so, it always returns a response. Each process P_i, $i \in I$, upon receiving an $init(v)$ input, invokes the unique k'-consensus service to which it is connected, with the same input v. It waits to receive a response, and then returns the same response to its own environment, using a $decide()$ output action. Since the implementation uses only $g = k/k'$ k'-consensus services, this yields at most k distinct responses overall. The implementation of n-endpoint k-set-consensus is wait-free, i.e., it tolerates up to $n - 1$ faults.

To be concrete, suppose that n is an arbitrary even number, $n' = n/2$, $k = 2$, $k' = 1$, $f = n - 1$, and $f' = n/2 - 1$. Then this construction shows that wait-free n-endpoint 2-set consensus can be implemented from wait-free $n/2$-endpoint consensus services.

5 Impossibility of Boosting for Failure-Oblivious Services

A *failure-oblivious* service is a generalization of an atomic object. It allows a *perform* step to depend on which endpoint's $inv-buffer$ is being serviced. It allows a *perform* step to place any number of responses in any subset of the $resp-buffer$s, instead of just one response in the $resp-buffer$ corresponding to the endpoint of the invocation. It also allows spontaneous *compute* steps, not triggered by a message in an $inv-buffer$; these may also place any number of responses in any $resp-buffer$s. The key constraint is that no step may depend on explicit knowledge of failure events.

In this section, we define the class of failure-oblivious services, give an example (totally ordered broadcast), and show how Theorem 2 can be extended to failure-oblivious services.

5.1 f-resilient failure-oblivious services

As for atomic objects, we begin by defining a canonical f-resilient failure-oblivious service.

A *canonical f-resilient failure-oblivious service* is parameterized by J, f, and k, which have the same meanings as in canonical atomic objects. However, in place of the sequential type parameter $\mathcal{T}$, the service has a *service type parameter* $\mathcal{U}$, which is a tuple $\langle V, V_0, invs, resps, glob, \delta_1, \delta_2 \rangle$. Here, V and V_0 are as before, *invs* and *resps* are the respective sets of invocations and responses (which can occur at any endpoint), *glob* is a set of *global task names*, and δ_1 and δ_2 are transition relations. A global task is used to perform computation that involves invocations from and responses to several processes. For example, in the totally ordered broadcast example presented in Section 5.2 below, the $compute_{g,k}$ internal action (see Figure 7) takes the first message in an internal queue (*msgs*) and places it onto the response buffer of *every* process that is connected to the totally ordered broadcast service. This cannot be done by a $perform_{i,k}$ action, which can only access the invocation and response queues for a single process, namely P_i.

Letting *ResponseMap* denote the set of mappings from endpoint set J to the set of finite sequences of *resps*, we assume:

- δ_1 is a total binary relation from $invs \times J \times V$ to $ResponseMap \times V$.
  It is used in *perform* steps to map an invocation at the head of a particular $inv-buffer$, and the current value for *val*, to a set of possible results, each of which consists of a new value for *val* and finite sequences of responses to be added to the $resp-buffer$s.

- δ_2 is a total binary relation from $glob \times V$ to $ResponseMap \times V$.
  It is used in *compute* steps to map a value of *val* to a set of possible results, each of which again consists of a new value for *val* and finite sequences of responses to be added to the $resp-buffer$s.

The code for a canonical failure-oblivious automaton, showing specifically how these parameters are used, appears in Figure 4. Note (in the *dummy_compute* transition definition) that global tasks are allowed to stop performing steps when either the total number of failures exceeds f, or all of the endpoints have failed.

Thus, a canonical f-resilient failure-oblivious service is allowed to perform rather flexible kinds of processing, as long as processing decisions do not depend on knowledge of occurrence of failure events.

Notice that the canonical atomic object **CanonicalAtomicObject**$(\mathcal{T}, J, f, k)$ is a special case of the canonical failure-oblivious service **CanonicalFailureObliviousService**$(\mathcal{U}, J, f, k)$: In this special case, the J, f, and k parameters are the same. For a given sequential type $\mathcal{T} = \langle V, V_0, invs, resps, \delta \rangle$, the corresponding service type $\mathcal{U}$ is defined as $\langle V, V_0, invs, resps, glob, \delta_1, \delta_2 \rangle$, where $glob = \emptyset$, δ_2 is the empty relation, and δ_1 is defined as follows: δ_1 is the set of pairs $((a, i, v), (B, v'))$ for which there exists $b \in resps$ such that $((a, v), (b, v')) \in \delta$, $B(i)$ is the sequence consisting of a single b, and $B(j)$ is the empty sequence for every $j \neq i$.

An I/O automaton A is an *f-resilient failure-oblivious service* of type $\mathcal{U}$ for endpoint set J and index k, provided that it implements the canonical f-resilient failure oblivious service of type $\mathcal{U}$ for J and k, where "implements" is defined as in Section 2.1.1. It follows that an *f-resilient atomic object* of sequential type $\mathcal{T}$ for endpoint set J and index k is in fact an f-resilient failure-oblivious service of service type $\mathcal{U}$ for endpoint set J and index k, where the type $\mathcal{U}$ is derived from the type $\mathcal{T}$ as described just above.

5.2 Example: Totally Ordered Broadcast

Here we describe an f-resilient totally ordered broadcast service for a particular message alphabet M, endpoint set J, and index k, as a special case of an f-resilient failure-oblivious service for J and k. To do this, we need only specify the failure-oblivious service type $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2 \rangle$. Here, V consists of a single *msgs* queue, containing messages that have been totally ordered, together with their sources (Figure 5). V_0 indicates that this queue is initially empty.

The invocation set *invs* is $\{bcast(m) : m \in M\}$. The response set *resps* is $\{rcv(m, i) : m \in M, i \in J\}$. Here, $rcv(m, i)$ indicates the receipt of message m from sender i. This receipt can occur at any endpoint. *glob* consists of one task name g, that is, $glob = \{g\}$.

δ_1, the relation describing the transitions that process invocations from $inv-buffer$s, is defined implicitly in Figure 6. This code processes the first element of $inv-buffer(i)$ by adding it to the

CanonicalFailureObliviousService$(\mathcal{U}, J, f, k)$, where $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2 \rangle$

Signature:

Inputs:
$a_{i,k}$, $a \in invs$, $i \in J$
$fail_i$, $i \in J$

Outputs:
$b_{i,k}$, $b \in resps$, $i \in J$

Internals:
$perform_{i,k}$, $i \in J$
$compute_{g,k}$, $g \in glob$
$dummy\_perform_{i,k}$, $i \in J$
$dummy\_compute_{g,k}$, $g \in glob$
$dummy\_output_{i,k}$, $i \in J$

Tasks:
For every $i \in J$:
    i-perform: $\{perform_{i,k}, dummy\_perform_{i,k}\}$
    i-output: $\{b_{i,k} : b \in resps\} \cup \{dummy\_output_{i,k}\}$
For every $g \in glob$:
    g-compute: $\{compute_{g,k}, dummy\_compute_{g,k}\}$

State components:
As for canonical atomic object.

Transitions:

Input: $a_{i,k}$
As for canonical atomic object.

```
Internal: $perform_{i,k}$
Precondition:
    $a = head(inv-buffer(i))$
    $\delta_1((a, i, val), (B, v))$
Effect:
    remove head of $inv-buffer(i)$
    $val \leftarrow v$
    for $j \in J$ do
        add $B(j)$ to end of $resp-buffer(j)$
```

```
Internal: $compute_{g,k}$, $g \in glob$
Precondition:
    $\delta_2((g, val), (B, v))$
Effect:
    $val \leftarrow v$
    for $j \in J$ do
        add $B(j)$ to end of $resp-buffer(j)$
```

Output: $b_{i,k}$
As for canonical atomic object.

Input: $fail_i$
As for canonical atomic object.

Internal: $dummy\_perform_{i,k}$, $i \in J$
As for canonical atomic object.

```
Internal: $dummy\_compute_{g,k}$, $g \in glob$
Precondition:
    $|failed| > f \vee failed = J$
Effect:
    none
```

Internal: $dummy\_output_{i,k}$, $i \in J$
As for canonical atomic object.

Figure 4: A canonical failure-oblivious service.

Components of *val*:
$msgs$, a finite sequence of items in $M \times J$, initially empty

Figure 5: Components of val in a totally ordered broadcast service.

end of the sequence stored in *msgs*. Formally, $\delta_1((a,i,v),(B,v'))$ holds if and only if $a = bcast(m)$, $v'.msgs$ is the result of adding (m,i) to the end of $v.msgs$, and $B(j)$ is empty for all j.

```
**Internal:** $perform_{i,k}$
Precondition:
    $bcast(m) = head(inv-buffer(i))$
Effect:
    remove head of $inv-buffer(i)$
    add $(m,i)$ to end of $msgs$
```

Figure 6: Relation δ_1 in a totally ordered broadcast service.

Relation δ_2 is defined implicitly in Figure 7. This code processes the first element of *msgs* by removing it from *msgs* and adding it to the end of the sequence of messages stored in $resp-buffer(j)$, for every j. Formally, $\delta_2((g,v),(B,v'))$ holds if and only if either (a) $v.msgs$ is nonempty, $(m,i) = head(v.msgs)$, $v'.msgs = tail(v.msgs)$, and for every $j \in J$, $B(j)$ is the sequence consisting of the single element $rcv(m,i)$, or (b) $v.msgs$ is empty, $v' = v$, and for every j, $B(j)$ is the empty sequence.

Note that totally ordered broadcast cannot be implemented by an atomic object, since one invocation requires many responses. Thus the notion of failure-oblivious service increases the range of systems that our framework can express.

```
**Internal:** $compute_{g,k}$
Precondition:
    true
Effect:
    if $msgs$ is nonempty then
        $(m,i) := head(msgs)$
        remove head of $msgs$
        for $j \in J$ do
            add $rcv(m,i)$ to end of $resp-buffer(j)$
```

Figure 7: Relation δ_2 in a totally ordered broadcast service.

5.3 Impossibility of Boosting

Now we show that Theorem 2 extends to the case of failure-oblivious services. The new theorem is:

Theorem 9 *Let I be a set of endpoints, $n = |I|$, and let f be an integer such that $0 \le f < n-1$. There is no distributed system using only canonical f-resilient failure-oblivious services and canonical reliable registers that solves $(f+1)$-resilient binary consensus for I.*

Proof: The proof follows the same outline as for Theorem 2. We sketch the modifications here. First, the index set K is now the set of indices of all the f-resilient failure-oblivious services in $\mathcal{C}$. The f-resilient failure-oblivious service with index k has a service type $\mathcal{U}_k = \langle V_k, (V_0)_k, invs_k, resps_k, glob_k, (\delta_1)_k, (\delta_2)_k \rangle$,

Lemma 1 extends to this case because the only relevant modification to the service is the addition of the g-compute tasks. These are defined using the total transition relation δ_2. Since these are total relations, we see from Figure 4 that these tasks are always enabled. It follows that Lemma 1 still holds.

For determinism assumptions, we require the processes to be deterministic automata as before, and also require the transition relations δ_1 and δ_2 to be (single-valued) functions. Lemmas 3, 4, and 5 and their proofs carry over without change, since they do not depend on the definition of a service. The similarity definitions are the same as before, except that the services S_k are now failure-oblivious services instead of atomic objects.

For Lemmas 6, 7, and 8, we provide complete proofs in Appendix A. Here, we just sketch the changes. For Lemma 6, the execution fragments γ' and γ'' now may contain $compute_{g,k}$ actions. We argue that these do not invalidate the inductive argument that shows the correspondence between γ' and γ''.

For Lemma 7, the proof requires very little change. Service S_k performs no locally controlled actions, including $compute_{g,k}$ actions, in either γ' or γ'', and all other services and processes behave the same in γ' and γ''. The changes to the definition of S_k do not affect the proof, since the original proof of Lemma 7 does not depend on the detailed definitions of the services.

For Lemma 8, Claims 1, 2, 3, and 5 carry over with no change, since their proofs do not involve the details of the definitions of atomic objects or failure-oblivious services. For Claim 4, the proof of case 1 ($participants(e, s) = participants(e', s) = \{S_k\}$) is modified by considering g-compute tasks as well as i-perform tasks. The proofs of the other cases carry over directly. Hence the lemma as a whole carries over. □

6 Impossibility of Boosting for General (Failure-Aware) Services

A *general*, or (potentially) *failure-aware* service is a further generalization of a failure-oblivious service. As for failure-oblivious services, a general service has both *perform* and *compute* steps. The difference is that a general service does not have the failure-oblivious constraint: its decisions may depend on knowledge of past failures of processes connected to the service.

In this section, we define the class of general services, give examples, and show how Theorems 2 and 9 can be extended to such services. The extension is weaker than the previous theorems, in that it requires a constrained connection pattern: all processes must be connected to all general services. We show by example that this constraint is needed: without it, boosting is sometimes possible.

6.1 f-resilient general services

A *canonical f-resilient general service* is parameterized by J, f, and k, which have the same meanings as for canonical atomic objects and canonical failure-oblivious services, and by a service type parameter $\mathcal{U}$, which is a tuple of the form $\langle V, V_0, invs, resps, glob, \delta_1, \delta_2\rangle$, as for failure-oblivious services. This time, however:

- δ_1 is a total binary relation from $invs \times J \times V \times 2^I$ to $ResponseMap \times V$.
  As for failure-oblivious services, δ_1 is used in *perform* steps. The final argument, of type 2^I, is instantiated in the *perform* code with the current *failed* set.

- δ_2 is a total binary relation from $glob \times V \times 2^I$ to $ResponseMap \times V$.
  As for failure-oblivious services, δ_2 is used in *compute* steps. The final argument is again instantiated in the *compute* code with the current *failed* set.

We call the new service **CanonicalGeneralService**$(\mathcal{U}, J, f, k)$. The only portions of its code that differ from those for **CanonicalFailureObliviousService**$(\mathcal{U}, J, f, k)$ are the two transition definitions that use δ_1 and δ_2; the new ones appear in Figure 8.

```
**CanonicalGeneralService**$(\mathcal{U}, J, f, k)$, where $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2\rangle$

Internal: $perform_{i,k}$
Precondition:
    $a = head(inv-buffer(i))$
    $\delta_1((a, i, val, failed), (B, v))$
Effect:
    remove head of $inv-buffer(i)$
    $val \leftarrow v$
    for $j \in J$ do
        add $B(j)$ to end of $resp-buffer(j)$

Internal: $compute_{g,k}$, $g \in glob$
Precondition:
    $\delta_2((g, val, failed), (B, v))$
Effect:
    $val \leftarrow v$
    for $j \in J$ do
        add $B(j)$ to end of $resp-buffer(j)$
```

Figure 8: Relations δ_1 and δ_2 in a canonical f-resilient general service.

Notice that the canonical failure-oblivious service **CanonicalFailureObliviousService**$(\mathcal{U}, J, f, k)$ is a special case of the canonical general service **CanonicalGeneralService**$(\mathcal{U}', J, f, k)$, with the same J, f, and k parameters. For a given service type $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2\rangle$ for a canonical failure-oblivious service, the corresponding service type $\mathcal{U}'$ for a canonical general service is defined as $\langle V, V_0, invs, resps, glob, \delta_1', \delta_2'\rangle$, where δ_1' is the set of pairs $((a, i, v, F), (B, v'))$ such that $((a, i, v), (B, v')) \in \delta_1$, and δ_2' is the set of pairs $((g, v, F), (B, v'))$ such that $((g, v), (B, v')) \in \delta_2$,

An I/O automaton A is an *f-resilient general service* of type $\mathcal{U}$ for endpoint set J and index k, provided that it implements the canonical f-resilient general service of type $\mathcal{U}$ for J and k.

6.2 Examples: Failure detectors

In this section, we describe how two well-known failure detectors [4, 5] can be modeled as general services. Our failure detectors do not provide all the functionality of the standard model [4]; most notably, because our failure detectors are modeled as automata, they cannot predict future input actions and their output can only depend on the order in which failures take place, and not on the timing of failures. We conjecture that our framework allows for describing the subset of realistic failure detectors [7] that are "time-independent," i.e., depend only on the relative order of failures.

All of our failure detector services have empty *invs* sets, that is, their only inputs are $fail_i$ actions.

6.2.1 Perfect Failure Detector $\mathcal{P}$

A perfect failure detector is supposed to provide all its endpoints with recent, accurate information about which endpoints have failed. We define an f-resilient perfect failure detector for J and k as a canonical f-resilient general service of type $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2\rangle$. Here, V contains only one (trivial) state $\bar{v}$, that is, the service maintains no internal information other than the *failed* set. As noted above, $invs = \emptyset$. Responses are of the form $suspect(J')$, $J' \subseteq J$. The set *glob* of global task names is simply J—that is, it contains exactly one task name for each endpoint in J. Since there are no invocations, δ_1 is trivial.

It remains to define δ_2, which describes the generation of *suspect* responses for particular endpoints. $\delta_2(i, \bar{v}, failed)$ simply puts a *suspect* response containing the current *failed* set into i's response buffer. Formally, $\delta_2((i, \bar{v}, failed), (B, \bar{v})$ holds iff $B(i)$ is the sequence consisting of the single element *suspect*(*failed*) and for every $j \in J - \{i\}$, $B(j)$ is the empty sequence. Figure 9 shows *compute* code that uses δ_2 implicitly.

```
Internal:  $compute_{i,k}$
Precondition:
    true
Effect:
    add $suspect(failed)$ to $resp\!-\!buffer(i)$
```

Figure 9: Relation δ_2 in $\mathcal{P}$.

6.2.2 Eventually Perfect Failure Detector $\Diamond\mathcal{P}$

An eventually perfect failure detector is supposed to provide all its endpoints with information about which endpoints have failed. This information may be erroneous for some finite amount of time, but eventually it is supposed to stabilize so that thereafter, it is recent and accurate. We define an f-resilient eventually perfect failure detector for J and k, as a canonical f-resilient general service of type $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2 \rangle$. Here, V consists of valuations for a single *mode* variable, which takes on values in $\{perfect, imperfect\}$. V_0 assigns *mode* the value *imperfect*. Figure 10 contains an implicit definition of V and V_0.

```
Components of val:
    $mode \in \{perfect, imperfect\}$, initially $imperfect$
```

Figure 10: The components of *val* in $\Diamond\mathcal{P}$.

As before, $invs = \emptyset$. Responses are of the form $suspect(J')$, $J' \subseteq J$. Since there are no invocations, δ_1 is trivial. Now $glob = J \cup \{g\}$, so we have one global task name per endpoint plus one special task name g. Global task i, $i \in J$, is responsible for generating *suspect* responses for endpoint i, while global task g is a background task that is responsible for eventually switching *mode* to *perfect*. While *mode* is *imperfect*, the service may generate arbitrary *suspect* responses; after *mode* becomes *perfect*, the responses must be recent and accurate.

We define δ_2 implicitly, in the *compute* transition definitions in Figure 11.

```
Internal:  $compute_{i,k}$, $i \in J$
Precondition:
    true
Effect:
    if $mode = perfect$ then
        add $suspect(failed)$ to $resp\!-\!buffer(i)$
    else
        choose $J'$ where $J' \subseteq J$
        add $suspect(J')$ to $resp\!-\!buffer(i)$

Internal:  $compute_{g,k}$
Precondition:
    true
Effect:
    $mode := perfect$
```

Figure 11: Internal transitions in $\Diamond\mathcal{P}$.

6.3 Impossibility of Boosting

Our impossibility results for atomic objects and failure-oblivious services allow arbitrary connections between processes and services. However, it turns out that it *is possible* to boost the resilience of systems containing failure-aware services, if we allow arbitrary connection patterns.

For example, consider an n-process system that uses wait-free registers and 1-resilient canonical perfect failure detectors. Suppose that every pair of processes share a 1-resilient 2-process failure detector. Such a system can implement a *wait-free* perfect failure detector for all n processes as follows: Process i just listens to all failure detectors it is connected to and accumulates the set of suspected processes in a dedicated register. Periodically, it reads these dedicated registers and outputs the union of all sets of suspected processes. By the definition of 1-resilient 2-process perfect failure detectors, the 2-process services continually provide each process with accurate failure information about each other process. Therefore, the algorithm allows each process to continually provide accurate failure information about all n processes, as required by the definition of a wait-free n-process perfect failure detector. Using this construction, f-resilient consensus, for any f, can be implemented using wait-free registers and 1-resilient failure detector services.

Boosting is, however, impossible if we assume a system in which f-resilient failure-aware services must be connected to all processes, and so, any $f+1$ process failures can disable all the failure-aware services.

We obtain the following theorem; notice that we allow f-resilient failure-oblivious services, each connected to an *arbitrary* set of processes, in addition to general services connected to all processes.

Theorem 10 *Let I be a set of endpoints, $n = |I|$, and let f be an integer such that $0 \leq f < n-1$. There is no distributed system using only (a) canonical f-resilient general services connected to all processes, (b) canonical f-resilient failure-oblivious services (connected to arbitrary processes), and (c) canonical reliable registers (connected to arbitrary processes) that solves $(f+1)$-resilient binary consensus for I.*

Proof: The proof follows the same outline as for Theorem 9, based on similarity and the "hook" construction. The key new fact is that, when we fail $f+1$ processes in the proof of Lemma 6 or 7, we can "silence" all the failure-aware services.

The index set K is now partitioned into $K_1 \cup K_2$, where K_1 is the set of indices of all the f-resilient failure-oblivious services and K_2 is the set of indices of the f-resilient general services. Lemma 1 extends easily. For determinism assumptions, we require the processes to be deterministic automata, and also require the transition relations δ_1 and δ_2 to be (single-valued) functions. Lemmas 3, 4, and 5, and their proofs carry over without change from Section 5.3.

The similarity definitions now change so that they do not restrict the states of failure-aware services, that is, failure-aware services can have arbitrary states in s_0 and s_1. More precisely, let $j \in I$ and let s_0 and s_1 be states of $\mathcal{C}$. Then s_0 and s_1 are *j-similar* if the following hold:

(1) For every $i \in I - \{j\}$, the state of P_i is the same in s_0 and s_1.

(2) For every $c \in K_1 \cup R$:

1. The value of val_c is the same in s_0 and s_1.
2. For every $i \in J_c - \{j\}$, the value of $buffer(i)_c$ is the same in s_0 and s_1.

Note that we do not restrict the states for $c \in K_2$, that is, for the general services.

Also, let $k \in K$, and let s_0 and s_1 be states of $\mathcal{C}$. Then s_0 and s_1 are *k-similar* if the following hold:

(1) For every $i \in I$, the state of P_i is the same in s_0 and s_1.

(2) For every $c \in (K_1 - \{k\}) \cup R$, the state of S_c is the same in s_0 and s_1.

Again, we do not restrict the states for $c \in K_2$, that is, general services. For $k \in K_1$, this definition implies that all failure-oblivious services except for k have the same state in s_0 and s_1. For $k \in K_2$, this definition implies that all failure-oblivious services have the same state in s_0 and s_1.

Lemma 6 is stated as before, and the proof requires only small modifications to the corresponding proof for the failure-oblivious case. Now when we fail the $f+1$ processes in J in β, in addition to the other constraints on β, we require every failure-aware service to stop performing (non-dummy) locally controlled steps; we can do this because the $f+1$ failed processes are all connected to every failure-aware service. Then, following the strategy in the proof of Lemma 6 for the failure-oblivious case, we construct a failure-free extension γ' of α_0, $\alpha_0 \cdot \gamma'$, such that: (1) γ' includes no output actions of process P_i, nor any $perform_i$ or $output_i$ actions for any service, for $i \in J$, (2) γ' includes no locally controlled actions of any failure-aware service, and (3) γ' includes $decide(v)_l$, for some $l \in I - J$. Then we show that (essentially the same) γ' can be appended to α_1, which contradicts the assumption that α_0 and α_1 have opposite valences. In showing that γ' can be appended to α_1, we use arguments like those in proof of Lemma 6 for the failure-oblivious case. Since γ' contains no locally controlled actions of any failure-aware services, the new definitions for the *perform* and *compute* steps, in particular, their ability to observe the set of failed processes, make no difference.

Lemma 7 is also stated as before, although now the set K mentioned in the lemma is the union $K = K_1 \cup K_2$ of indices of failure-oblivious and general services. In the proof, when we fail the $f+1$ processes in J in β, we also require every failure-aware service to stop performing (non-dummy) locally controlled steps. Then, following the strategy in the proof of Lemma 7 for the failure-oblivious case, we construct a failure-free extension γ' of α_0, $\alpha_0 \cdot \gamma'$, such that: (1) γ' includes no locally controlled actions of service S_k, (2) γ' includes no locally controlled actions of any failure-aware service, and (3) γ' includes $decide(v)_l$, for some $l \in I - J$. Then we show that γ' is essentially applicable to α_1, which contradicts the assumption that α_0 and α_1 have opposite valences. In showing that γ' is applicable to α_1, we use arguments like those in the proof of Lemma 7 for the failure-oblivious case. Again, since γ' contains no locally controlled actions of any failure-aware services, the new definitions for the *perform* and *compute* steps make no difference.

For Lemma 8, note that none of the executions comprising the hook contain any $fail_i$ actions. Hence at all states in the hook, the set *failed* of failed processes is empty. Thus, the new definitions for the *perform* and *compute* steps, in particular, their ability to observe the set of failed processes, makes no difference. Hence the proof is unchanged from that for failure-oblivious services. $\square$

7 Conclusions

In this paper, we have presented a new framework for describing asynchronous distributed systems that use resilient services to implement other resilient services. The framework is general enough to describe atomic objects, other failure-oblivious services, and failure-aware services. To our knowledge, this is the first framework that can describe all of these.

Within our framework, we have established the impossibility of boosting the resilience of services. Specifically, we proved that f-resilient atomic objects, and more generally, f-resilient failure-oblivious services, cannot be used to solve $(f+1)$-resilient consensus. This is so even if processes can be connected to services using an arbitrary connection pattern. We have also proved that f-resilient failure-aware services cannot be used to solve $(f+1)$-resilient consensus; however, this proof, and in fact this result, require that all processes be connected to all failure-aware services. Our results can be viewed as generalizations, to any number f of failures, of the impossibility result of Fischer, Lynch, and Paterson [8] for the case $f = 0$. We emphasize that the result of Fischer, Lynch, and Paterson [8] does not imply our results, since the model of distributed systems used in [8] does not contain fault-tolerant services, and so the issue of boosting an existing nontrivial ($f > 0$) degree of fault-tolerance cannot be directly addressed in the framework of [8].

Our proofs are short, simple, and self-contained. They use techniques inspired by those in [8], in particular, bivalence and a "hook" construction. As a new addition to the proof method, we extract notions of "similarity" for processes and services, which describe relationships between states that ensure that they must lead to the same decision value. Our lemmas about similarity encapsulate reasoning about executions and valence, so that the main proof can focus exclusively on what happens in a few individual steps.

The consensus problem is a natural benchmark to use for measuring the resilience of services because it has already been shown to be fundamental to the study of resilience in distributed systems. In fact, this choice is crucial because our non-boosting results do not apply to some problems that are weaker than consensus, such as k-set-consensus.

As observed in [9,13], a variant of Theorem 2, for atomic objects, can be derived indirectly, by a chain of existing results in the literature [3,11,14]. However, these models differ in some technical aspects. Our self-contained proof is simpler than the indirect proof, if we take into account the complexity of the proofs of the constituent pieces. Our proof also extends readily to more general services than atomic objects. Our results for more general services are the first such results to appear.

This paper suggests several directions for future work. First, we have classified services in a hierarchy, as atomic objects, failure-oblivious services, and general (possibly failure-aware) services. Are there interesting refinements to this hierarchy? In particular, are there any interesting service classifications between failure-oblivious services and general services? If so, what boosting results apply to these services?

Also, our framework can be used to address the general question of which services can be used to implement which other services, with which levels of resilience. Thus, it could be used as the foundation for a general theory of relative computability of resilient services. Some results that would fit into such a theory already appear in the literature (e.g., [3,11]). It remains to develop a more complete theory.

References

[1] P. Attie, R. Guerraoui, P. Kouznetsov, N. A. Lynch, and S. Rajsbaum. The impossibility of boosting distributed service resilience. In *The 25'th International Conference on Distributed Computing Systems*, 2005.

[2] P. C. Attie, N. A. Lynch, and S. Rajsbaum. Boosting fault-tolerance in asynchronous message passing systems is impossible. Technical report, MIT Laboratory for Computer Science, MIT-LCS-TR-877, 2002. Available at `http://theory.lcs.mit.edu/tds/reflist.html`.

[3] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus. *SIAM Journal on Computing*, 34(2):333–357, 2005. Conference version appears in PODC94.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[6] S. Chaudhuri. Agreement is harder than consensus: set consensus in totally asynchronous systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, pages 311–324, August 1990.

[7] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. A realistic look at failure detectors. In *IEEE Symposium on Dependable Systems and Networks (DSN 2002)*, Washington DC, June 2002.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(3):374–382, April 1985.

[9] R. Guerraoui and P. Kouznetsov. Failure detectors as type boosters. *Distributed Computing*, 20(5):343–358, 2008.

[10] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcast and related problems. Technical report, Cornell University, Computer Science, May 1994.

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

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

[13] P. Jayanti. Private communication. 2003.

[14] P. Jayanti and S. Toueg. Some results on the impossibility, universability and decidability of consensus. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92)*, volume 647 of *LNCS*. Springer Verlag, 1992.

[15] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.

[16] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[17] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[18] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.

Appendix A Detailed proofs of main lemmas for failure-oblivious services

We present the proof of Lemma 6 for f-resilient failure-oblivious services.

Lemma 6 Let $j \in I$. Let α_0 and α_1 be finite failure-free input-first executions, s_0 and s_1 the respective final states of α_0 and α_1. Suppose that s_0 and s_1 are j-similar. If α_0 and α_1 are univalent, then they have the same valence.

Proof: We proceed by contradiction. Fix j, α_0, α_1, s_0, and s_1 as in the hypotheses of the lemma, and suppose (without loss of generality) that α_0 is 0-valent and α_1 is 1-valent. Let $J \subseteq I$ be any set of indices such that $j \in J$ and $|J| = f + 1$. Since $f < n - 1$ by assumption, we have $|J| < n$, and so $I - J$ is nonempty.

Consider a fair extension of α_0, $\alpha_0 \cdot \beta$, in which the first $f+1$ actions of β are $fail_i$, $i \in J$, and no other *fail* actions occur in β. Note that, for every $i \in J$, β contains no output actions of P_i. Assume that in β, no $perform_{i,c}$ or $b_{i,c}$ action of any $i \in J$ occurs at any component $c \in K \cup R$; we may assume this because, for each $i \in J$, action $fail_i$ enables a *dummy* action in every i-perform and i-output task of every service and register. Note that $compute_{g,k}$ and $dummy\_compute_{g,k}$ actions may occur in β, for $k \in K$.

Since α_0 is a failure-free input-first execution, the resulting extension $\alpha_0 \cdot \beta$ is a fair input-first execution containing $f + 1$ failures. Therefore, by the termination property for $(f + 1)$-resilient consensus and the fact that $I - J$ is nonempty, there is a finite prefix of $\alpha_0 \cdot \beta$, which we denote by $\alpha_0 \cdot \gamma$, that includes $\mathit{decide}(v)_l$ for some $l \notin J$ and $v \in \{0, 1\}$. Construct $\alpha_0 \cdot \gamma'$, where γ' is obtained from γ by removing the fail_i action and all subsequent internal actions of P_i, for every $i \in J$, plus all *dummy* actions, The *dummy* actions removed include the $\mathit{dummy\_compute}_{g,k}$ actions, as well as the $\mathit{dummy\_perform}_{i,k}$ and $\mathit{dummy\_output}_{i,k}$ actions. Thus, $\alpha_0 \cdot \gamma'$ is a failure-free extension of α_0 that includes $\mathit{decide}(v)_l$. Since α_0 is 0-valent, v must be equal to 0.

Note that $\mathit{compute}_{g,k}$ actions, but not $\mathit{dummy\_compute}_{g,k}$ actions, may occur in γ', for $k \in K$.

Next, we claim that $\mathit{decide}(0)_l$ occurs in the suffix γ', rather than in the prefix α_0. The argument is exactly as before.

Now we show how to append essentially the same γ' after α_1. The definition of j-similarity and the fact that $j \in J$ imply that:

(a) For every $i \notin J$, the state of P_i is the same in s_0 and s_1.

(b) For every $c \in K \cup R$,

1. The value of val_c is the same in s_0 and s_1 (that is, in the final states of α_0 and α_1).
2. For every $i \in J_c - J$, the value of $\mathit{buffer}(i)_c$ is the same in s_0 and s_1.

We know that, for every $i \in J$, γ' contains no locally controlled actions of P_i, and contains no $\mathit{perform}_{i,c}$ or $b_{i,c}$ actions, for any $c \in K \cup R$. Therefore:

(c) If γ' contains any locally controlled actions of a process P_i, then the state of P_i is the same in s_0 and s_1 (since $i \notin J$ in this case).

(d) For every $c \in K \cup R$,

1. The value of val_c is the same in s_0 and s_1.
2. For every $i \in J_c$, if γ' contains any $\mathit{perform}_{i,c}$ or $b_{i,c}$ ($b \in \mathit{resps}_c$) actions, then the value of $\mathit{buffer}(i)_c$ is the same in s_0 and s_1 (since $i \notin J$ in this case).

Then it is possible to append "essentially" the same γ' after α_1, resulting in a failure-free extension of α_1 that includes $\mathit{decide}(0)_l$. Since α_1 is 1-valent, this is a contradiction.

More precisely, we append another execution fragment γ'' after α_1—the one that is generated by applying, after α_1, the same sequence ρ of tasks that generates γ'. We can prove, by induction on the number of tasks, that when ρ is applied after α_0 and α_1, in each pair of corresponding states: (a) for each $i \in I$ for which the unique task of P_i occurs in ρ, the states of P_i are the same, (b) for each $c \in K \cup R$, the values of val_c are the same, and (c) for each $c \in K \cup R$ and each $i \in J_c$ such that an i-perform or i-output task of S_c occurs in ρ, the values of $\mathit{buffer}(i)_c$ are the same. This correspondence is enough to imply that γ'' also includes the required $\mathit{decide}(0)_l$ action.

Notice that the possible presence of g-compute tasks in ρ does not invalidate the inductive argument. Since $\alpha_1 \cdot \gamma''$ contains no failures, each g-compute task of each failure-oblivious service S_k is always applicable, and its results (new val_k and sequences to append to the $\mathit{resp-buffer}$s) depend only on val_k. This is enough to preserve properties (a)-(c) above. □

Next, we present the proof of Lemma 7 for failure-oblivious services.

Lemma 7 Let $k \in K$. Let α_0 and α_1 be finite failure-free input-first executions, s_0 and s_1 the respective final states of α_0 and α_1. Suppose that s_0 and s_1 are k-similar. If α_0 and α_1 are univalent, then they have the same valence.

Proof: We proceed by contradiction. Fix k, α_0, α_1, s_0, and s_1 as in the hypotheses of the lemma, and suppose (without loss of generality) that α_0 is 0-valent and α_1 is 1-valent. Let $J \subseteq I$ be any set of indices such that $|J| = f+1$, and, if $|J_k| \leq f+1$, then $J_k \subseteq J$, whereas if $|J_k| > f+1$, then $J \subseteq J_k$.

Consider a fair extension of α_0, $\alpha_0 \cdot \beta$, in which the first $f+1$ actions of β are fail_i, $i \in J$, and no other *fail* actions occur in β. Note that, for every $i \in J$, β contains no output actions of P_i. Assume that in β, no $\mathit{perform}_{i,k}$ action, or $b_{i,k}$ action ($b \in \mathit{resps}_k$), or $\mathit{compute}_{g,k}$ ($g \in \mathit{glob}_k$) occurs; we may assume this because the $f+1$ *fail* actions enable *dummy* actions in all of the tasks of S_k.

Since α_0 is a failure-free input-first execution, the resulting extension $\alpha_0 \cdot \beta$ is a fair input-first execution containing $f+1$ fail actions. Therefore, by the termination property for $(f+1)$-resilient consensus and the fact that $I - J$ is nonempty, there is a finite prefix of $\alpha_0 \cdot \beta$, which we denote by $\alpha_0 \cdot \gamma$, that includes $\mathit{decide}(v)_l$ for some $l \notin J$ and $v \in \{0,1\}$. Construct $\alpha_0 \cdot \gamma'$, where γ' is obtained from γ by removing the fail_i action and subsequent internal actions of P_i, for every $i \in J$, plus all *dummy* actions. Thus, $\alpha_0 \cdot \gamma'$ is a failure-free extension of α_0 that includes $\mathit{decide}(v)_l$. Since α_0 is 0-valent, v must be equal to 0.

Next, we claim that $\mathit{decide}(0)_l$ occurs in the suffix γ, rather than in the prefix α_0. The argument is exactly as before.

Now we show how to append essentially the same γ' after α_1. The definition of k-similarity implies that:

(a) For every $i \in I$, the state of P_i is the same in s_0 and s_1.

(b) For every $c \in (K - \{k\}) \cup R$, the state of S_c is the same in s_0 and s_1.

We know that γ' contains no locally controlled actions of service S_k. Therefore:

(c) For every $c \in K \cup R$, if γ' contains any $\mathit{perform}_{i,c}$ or $b_{i,c}$ or $\mathit{compute}_{g,c}$ actions of S_c, then the state of S_c is the same in s_0 and s_1 (since $c \neq k$ in this case).

By properties (a) and (c), it follows that it is possible to append "essentially" the same γ' after α_1 (differing only in the state of S_k) resulting in a failure-free extension of α_1 that includes $\mathit{decide}(0)_l$. But α_1 is 1-valent — a contradiction. □

Finally, we present the proof of Lemma 8 for failure-oblivious services.

Lemma 8 $G(\mathcal{C})$ contains no hooks.

Proof: We establish the same five claims as in the case of atomic objects, which establishes the needed contradiction.

Claims 1, 2, and 5 do not refer to the definition of an atomic object or failure-oblivious service, and so their proof remains unchanged from the atomic objects case.

The proof of Claim 3 is also unchanged, since the only actions considered here have as participants either just a process P_i, or else both a process P_i and a service S_c, $c \in K \cup R$. Thus, whenever a failure-oblivious service S_k is a participant, the action must be an external action of S_k. Since the external actions in the definitions of atomic object and failure-oblivious service have the same effect, namely to add or remove a single item from a single buffer, it follows that the proof of Claim 3 for the atomic object case still applies.

We modify the proof of Claim 4 as follows:

Claim 4: There does not exist $k \in K$ such that $S_k \in \mathit{participants}(e,s) \cap \mathit{participants}(e',s)$. Suppose for contradiction that $S_k \in \mathit{participants}(e,s) \cap \mathit{participants}(e',s)$. There are four possibilities:

1. $\mathit{participants}(e, s) = \mathit{participants}(e', s) = \{S_k\}$. Then e and e' must be i-perform or g-compute tasks of S_k, and so involve only the state of S_k. But then the states s_0 and s_1 can differ only in the state of S_k. So s_0 and s_1 are k-similar — a contradiction.

2. For some $i \in I$, $\mathit{participants}(e, s) = \{S_k, P_i\}$ and $\mathit{participants}(e', s) = \{S_k\}$.
   Then $\mathit{action}(e, s)$ is either $a_{i,k}$ or $b_{i,k}$, and $\mathit{action}(e', s)$ is either $\mathit{perform}_{j,k}$ or $\mathit{compute}_{g,k}$, where $j \in J_k$ and $g \in \mathit{glob}_k$. Inspection of the definition of a canonical failure-oblivious service shows that the two tasks commute, that is, $e'(s_0) = s_1$ — a contradiction.

3. For some $i \in I$, $\mathit{participants}(e', s) = \{S_k, P_i\}$ and $\mathit{participants}(e, s) = \{S_k\}$.
   Then $\mathit{action}(e, s)$ is either $\mathit{perform}_{j,k}$ or $\mathit{compute}_{g,k}$, where $j \in J_k, g \in \mathit{glob}_k$, and $\mathit{action}(e', s)$ is either $a_{i,k}$ or $b_{i,k}$. Inspection of the definition of a canonical failure-oblivious service shows that the two tasks commute, that is, $e'(s_0) = s_1$ — a contradiction.

4. For some $i, j \in I$, $\mathit{participants}(e, s) = \{S_k, P_i\}$ and $\mathit{participants}(e', s) = \{S_k, P_j\}$.
   By Claim 3, we know that $i \neq j$. Then $\mathit{action}(e, s)$ is either $a_{i,k}$ or $b_{i,k}$, and $\mathit{action}(e', s)$ is either $a_{j,k}$ or $b_{j,k}$. Inspection of the definition of a canonical failure-oblivious service shows that the two tasks commute, that is, $e'(s_0) = s_1$ — a contradiction.

□

Appendix B Proof that our definition of consensus implies the axiomatic definition

We now show that any system that meets our definition of consensus also meets the variant of the axiomatic definition given in Section 2.2.4. We argue that the f-fault-tolerant canonical consensus object for endpoint set I satisfies the agreement, validity, and modified termination conditions given in Section 2.2.4.

Theorem 11 *Let S be an f-fault-tolerant canonical consensus object with endpoint set I. Then every execution of S in which at most one input arrives at each endpoint satisfies the agreement, validity, and modified termination conditions given in Section 2.2.4.*

Proof: Consider an arbitrary execution α of S. We show that α satisfies each of the agreement, validity, and modified termination conditions. Recall that an f-fault-tolerant canonical consensus object has the sequential type binary consensus given in Section 2.1.2.

Agreement condition. From the definition of the binary consensus sequential type, each endpoint in I has two invocations, $\mathit{init}(0)$, $\mathit{init}(1)$, and two responses, $\mathit{decide}(0)$, $\mathit{decide}(1)$. Also, the value of the f-fault-tolerant canonical consensus object is initially $\emptyset$, and on invocation $\mathit{init}(0)$ changes from $\emptyset$ to $\{0\}$, and on invocation $\mathit{init}(1)$ changes from $\emptyset$ to $\{1\}$, and is stable once it is different from $\emptyset$. It is also clear that any $\mathit{decide}(0)$ response is only issued by the consensus object when it has value $\{0\}$, and any $\mathit{decide}(1)$ response is only issued by the consensus object when it has value $\{1\}$. Hence, after the first $\mathit{decide}(0)$ response, all subsequent responses will be $\mathit{decide}(0)$, and after the first $\mathit{decide}(1)$ response, all subsequent responses will be $\mathit{decide}(1)$. So, α satisfies the agreement condition.

Validity condition. If all invocations are $\mathit{init}(0)$, then the only possible change of the consensus object is from $\emptyset$ to $\{0\}$. Hence, all responses will be $\mathit{decide}(0)$. Likewise if all invocations are $\mathit{init}(1)$, then all responses will be $\mathit{decide}(1)$. Otherwise, there are both $\mathit{init}(0)$ and $\mathit{init}(1)$ invocations.

Hence, in all cases, the value decided on is the value occurring in some invocation. Hence, α satisfies the validity condition.

Modified termination condition. Assume that at most f endpoints fail along α, and that the scheduling along α is in accord with the I/O automata fairness assumption, as discussed in Section 2.2.3. Consider any endpoint i that does not fail. If an input occurs at endpoint i, then eventually a $\mathit{perform}_{i,k}$ occurs (where k is the index of S), followed by a $\mathit{decide}(v)_{i,k}$ at endpoint i. Hence α satisfies the modified termination condition.

Since α is an arbitrary execution of S, we conclude that every execution of S satisfies the agreement, validity, and modified termination conditions, as desired. □

We remind the reader that the modified termination condition is different from the traditional termination condition, which requires that all nonfaulty processes do have an initial value, and that they all eventually decide. Here, only the nonfaulty processes that receive an input will make a decision.