The Impossibility of Boosting Distributed Service Resilience *

Paul Attie[1,3] Rachid Guerraoui[2] Petr Kouznetsov[2] Nancy Lynch[3] Sergio Rajsbaum[4]

(1) College of Computer Science, Northeastern University
(2) Distributed Programming Laboratory, EPFL
(3) MIT Computer Science and Artificial Intelligence Laboratory
(4) Instituto de Matemáticas, Universidad Nacional Autónoma de México (UNAM)

Abstract

We prove two 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 $f+1$ undetectable process stopping failures. (A service is f-resilient if it is guaranteed to operate as long as no more than f of the processes connected to it fail.)

Our first theorem assumes that the given services are atomic objects*, and allows any connection pattern between processes and services. In contrast, we show that it is possible to boost the resilience of systems solving problems easier than consensus: the k-set consensus problem is solvable for $2k-1$ failures using 1-resilient consensus services. The first theorem and its proof generalize to the larger class of* failure-oblivious *services.*

Our second theorem allows the system to contain failure-aware *services, such as failure detectors, in addition to failure-oblivious services; however, it 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 if arbitrary patterns of connectivity are allowed between processes and failure-aware services: consensus is solvable for any number of failures using only 1-resilient 2-process perfect failure detectors.*

1 Introduction

We consider distributed systems consisting of asynchronously operating processes that coordinate using reliable multi-writer multi-reader registers and shared services. 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, 14], for example, atomic read-modify-write, queue, counter, test&set, and compare&swap objects. The consensus problem can also be defined as an atomic object.
- Concurrently-accessible data structures such as balanced trees.
- Broadcast services such as totally ordered broadcast [10].
- Failure detectors, which provide processes with hints 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, 6, and 7. 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 it fail.

A fundamental, general question in distributed computing theory is: "What problems can be solved by distributed systems, with what levels of resilience, using services of given types and levels of resilience?" In this paper, we expose a basic limitation on the achievable resilience, namely, that the resilience of a system cannot be "boosted" above that of its services. More specifically, we prove two 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.

*The first author was supported by the National Science Foundation under Grant No. 0204432

[1] Our notion of service encompasses all failure detectors defined by Chandra et al. [4] with one exception: we exclude failure detectors that can guess the future.

We focus on the consensus problem because it has been shown to be fundamental to the study of resilience in distributed systems. For example, Herlihy has shown that consensus is universal [11]: an atomic object of any sequential type can be implemented in a *wait-free* manner (i.e., tolerating any number of failures), using wait-free consensus objects.

Our first main theorem, Theorem 1, assumes that the given services are *atomic objects* and allows any connection pattern between processes and services. The result is a strict generalization of the classical impossibility result of Fischer et al. [8] for fault-tolerant consensus. Our simple, self-contained impossibility proof is based on a bivalence argument similar to 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 k-set consensus problem [6] is solvable for $2k-1$ failures using 1-resilient consensus services.

Theorem 1 and its proof assume that the given services are atomic objects; however, they extend to the larger class of *failure-oblivious* services. A failure-oblivious service generalizes an atomic object by allowing an invocation to trigger multiple processing steps instead of just one, and to trigger any number of responses, at any endpoints. The service may also include background processing tasks, not related to any specific endpoint. The key constraint is that no step may depend on explicit knowledge of failure events. We define the class of failure-oblivious services, give examples (e.g., totally-ordered broadcast), and describe how Theorem 1 can be extended to such services.

Our second main theorem, Theorem 11, addresses the case where the system may contain *failure-aware* services (e.g., failure detectors), 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 failure-aware service is connected to all processes; thus, $f+1$ process failures overall can disable all the failure-aware services. The proof is an extension of the first proof, using the same "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 are allowed between processes and failure-aware services: specifically, consensus is solvable for any number of failures using only 1-resilient 2-process perfect failure detectors.

Related work. Our Theorem 1, for atomic services, can be derived by carefully combining several earlier theorems, including Herlihy's result on universality of consensus [11], and the result of Chandra et al. on f-resiliency vs. wait-freedom [3] (see Appendix A). However, this argument does not extend to prove impossibility of boosting for failure-oblivious and failure-aware services. Moreover, some of the proofs upon which this alternative proof rests are themselves more complex than our direct proof.

Theorem 1 appeared first in a technical report [1]. Subsequent impossibility results for atomic objects appeared in [9, 15]. Our models for failure-oblivious services and general services are new. As far as we know, this is the first time a unified framework has been used to express atomic and non-atomic objects. Moreover, this is the first time boosting analysis has been performed for services more general than atomic objects.

Organization. Section 2 presents definitions for the underlying model of concurrent computation and for atomic objects. Section 3 presents our model for a system whose services are atomic objects. Section 4 presents the first impossibility result. Section 5 shows that boosting is possible for set consensus. Section 6 defines failure-oblivious services, gives an example, and extends the first impossibility result to systems with failure-oblivious services. Section 7 defines general services, gives examples, and presents our second main impossibility result. Appendix A shows how Theorem 1 can be derived from results in [3, 11] and why these arguments do not extend to services more general than atomic services. Appendix B provides the complete proofs for the extension of the first impossibility result to failure-oblivious services.

2 Mathematical Preliminaries

2.1 Model of concurrent computation

We use the I/O automaton model [18, chapter 8] as our underlying model for concurrent computation. We assume the terminology of [18, chapter 8]. An I/O automaton A is *deterministic* iff, 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.

2.2 Sequential types

We define the notion of a "sequential type", in order to describe allowable sequential behavior of atomic services. The definition used here generalizes the one in [18, 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 $((a, v), (b, v')) \in \delta$.

We sometimes use dot notation, writing $\mathcal{T}.V, \mathcal{T}.V_0, \mathcal{T}.\mathit{invs}, \ldots$ 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.

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 $\delta = \{((\mathit{read}, v), (v, v)) : v \in V\} \cup \{((\mathit{write}(v), v'), (\mathit{ack}, v)) : v, v' \in V\}$.

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 $\delta = \{((\mathit{init}(v), \emptyset), (\mathit{decide}(v), \{v\})) : v \in V\} \cup \{((\mathit{init}(v), \{v'\}), (\mathit{decide}(v'), \{v'\})) : v, v' \in V\}$

Example. k-consensus *sequential type:* Now V is the set of subsets of $\{0, 1, \ldots, k\}$ having at most k elements, $V_0 = \{\emptyset\}$, $\mathit{invs} = \{\mathit{init}(v) : v \in \{0, 1, \ldots, k\}\}$, $\mathit{resps} = \{\mathit{decide}(v) : v \in \{0, 1, \ldots, k\}\}$, 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.

2.3 Canonical f-resilient atomic objects

A "canonical f-resilient atomic object" describes the allowable concurrent behavior of atomic objects. Namely, we define the *canonical f-resilient atomic object of type $\mathcal{T}$ for endpoint set J and index k*, where

- $\mathcal{T}$ is a sequential type,
- J is a finite set of *endpoints* at which invocations and responses may occur,
- $f \in \mathbf{N}$ is the level of resilience, and
- k is a unique index (name) for the service.

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

The parameter J allows different objects to be connected to the same or different sets of processes. A process at endpoint $i \in J$ can issue any invocation specified by the underlying sequential type 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 sequential type $\mathcal{T}$, and due to interleavings of steps for different 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: For every process $i \in J$, we assume the service 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 i. In addition, every i-* task ($*$ is perform or output) contains a special $\mathit{dummy\_*}_{i,k}$ action, which is enabled when either process i has failed or more than f processes in J have failed. The $\mathit{dummy\_*}_{i,k}$ action is intended to allow, but not force, the service to stop performing

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:
$perform_{i,k}$, $i \in J$
$dummy\_*_{i,k}$, $* \in \{perform, output\}$, $i \in J$

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: $dummy\_*_{i,k}$
Precondition:
$i \in failed \vee |failed| > f \vee failed = J$
Effect:
none

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

Figure 1: A canonical atomic object.

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 $dummy\_*_{i,k}$ action every time the $i - *$ task gets a turn, which permits the object to avoid responding to i. In particular, if more than f processes fail, the object may avoid responding to any process in J, since $dummy\_ouput_{i,k}$ is enabled for all $i \in J$. Also, if all processes connected to the service (i.e., all processes in J) fail, the object may avoid responding to any process.

Thus, the basic fairness definition 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.

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

2.4 f-resilient atomic objects

An I/O automaton A is an *f-resilient atomic object* of type $\mathcal{T}$ for 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, in the following sense:

1. A and S have the same input actions (including *fail* actions) and the same output actions.
2. Any trace of A is also a trace of S. (This implies that A guarantees atomicity.)
3. Any fair trace of A is also a fair trace of S. (This says that A is f-resilient.)

We say that 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 $f \geq |J| - 1$.

3 System Model with Atomic Objects

Our system model consists of a collection of process automata, reliable registers, and fault-prone atomic objects (which we sometimes refer to as *services*). For this section, we fix I, K, and R, finite (disjoint) index sets for processes, services, and registers, respectively, and $\mathcal{T}$, a sequential type, representing the problem the system is intended to solve. A *distributed system* for I, K, R, and $\mathcal{T}$ is the composition of the following I/O automata (see [18, chapter 8]):

1. *Processes* P_i, $i \in I$,
2. *Services (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 service S_k. We assume k itself is the index.

3. *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 assume r is the index.

Processes interact only via services and registers. Process P_i can invoke an operation on service 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 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.

3.1 Processes

We assume that process P_i, $i \in I$ has the following 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 service S_k such that $i \in J_k$, outputs $a_{i,k}$, $a \in \mathcal{T}_k.invs$, and inputs $b_{i,k}$, $b \in \mathcal{T}_k.resps$.
- For every register S_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 $fail_i$.

P_i may issue several invocations, on the same or different services or registers, 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. As a technicality, we assume that when P_i performs a $decide(v)_i$ output action, it records the decision value v in a special state component.

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. We assume that the $fail_i$ input action affects P_i in such a way that, from that point onward, no output actions are enabled. However, other locally-controlled actions may be enabled—in fact, by the restriction just above, some such action *must* be enabled. This action might be a "dummy" action, as in the canonical resilient atomic objects defined in Section 2.3.

3.2 Services and registers

We assume that service S_k is the canonical f-resilient atomic object of type $\mathcal{T}_k$ for J_k and k. Likewise, 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.

3.3 The complete system

The complete system $\mathcal{C}$ is constructed by composing the P_i, S_k, and S_r automata and then hiding all the actions used to communicate among them.

For any action a of $\mathcal{C}$, we define the *participants* of action a to be the set of automata with a in their signature. Note that no two distinct registers or services participate in the same action a, and similarly no two distinct processes participate in the same action. Furthermore, for any action a, the number of participants is at most 2. Thus, if an action a has two participants, they must be a process and either a service or register.

As we defined earlier, each process P_i has a single task, consisting of all the locally controlled actions of P_i. Each service or register S_c, $c \in K \cup R$, has two tasks for each $i \in J_c$: i-perform, consisting of $\{perform_{i,k}, dummy\_perform_{i,k}\}$, and i-output, consisting of $\{b_{i,k} : b \in \mathcal{T}_k.resps\} \cup \{dummy\_output_{i,k}\}$. 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 assumptions imply that each of these tasks get 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.

3.4 The 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.[2] 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., [18, chapter 6]):

Agreement No two processes decide on different values.

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

[2]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.

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: We say that a distributed system S *solves f-resilient consensus for I* if and only if S is an f-resilient atomic object of type consensus (Section 2.2) for endpoint set I. We argue 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 before; our new termination condition is:

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

4 Impossibility of Boosting for Atomic Objects

Our first main theorem is:

Theorem 1 *Let $n = |I|$ be the number of processes, and let f be an integer such that $0 \leq f < n-1$. There does not exist an $(f+1)$-resilient n-process implementation of consensus from canonical f-resilient atomic objects and canonical reliable registers.*

To prove Theorem 1, 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, $i \in I$, services S_k, $k \in K$, and registers S_r, $r \in R$. By assumption, $\mathcal{C}$ satisfies the agreement, validity and termination properties of consensus.

For each component $c \in K \cup R$ and $i \in J_c$ (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 $buffer(i)_c = \langle inv-buffer(i)_c, resp-buffer(i)_c\rangle$.

4.1 Assumption

To prove Theorem 1, we make the following assumption:

(i) We assume that the processes P_i, $i \in I$, are deterministic automata, as defined in Section 2.1. For services, we assume a slightly weaker condition: that the sequential type is deterministic, i.e, the sequential type has a unique initial value and the transition relation δ is a mapping. Note that the sequential type for registers is also deterministic, by definition.

Assumption (i) implies 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 as $transition(e, s)$ (since it is uniquely defined by the final state s). If $transition(e, s) = (s, a, s')$, then we write $first(e, s)$, $action(e, s)$, and $last(e, s)$ to denote s, a, and s', respectively. We sometimes abbreviate $last(e, s)$ as $e(s)$. Note that, if s is the final state of α, then $transition(e, s)$, $first(e, s)$, $action(e, s)$, and $last(e, s)$ are defined iff e is applicable to α.

Assumption (i) implies that any *failure-free* execution can be defined by applying a sequence of tasks, one after the other, to the initial state of $\mathcal{C}$. Assumption (i) does not reduce the generality of our impossibility result, because any candidate system could be restricted to satisfy (i); if the impossibility result holds for the restricted automaton, then it also holds for the original one.

Lemma 2 *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 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, service task, 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_k, then applicability of e to α means that service S_k has either a pending invocation in an $inv-buffer$ or a pending response in a $resp-buffer$, 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 after $\alpha \cdot \beta$. If e is a register task, the argument is similar. □

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 $participants(e, s)$ for the set of participants of action $action(e, s)$. Note that, for any task e and any state s, $|participants(e, s)| \leq 2$. Also, if $|participants(e, s)| = 2$, then $participants(e, s)$ is of the form $\{P_i, S_c\}$, for some $i \in I$ and $c \in K \cup R$.

4.2 Initializations and valence

In our proof, we consider executions 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 $init()_i$ 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 *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 $decide(0)_i$ action, for some $i \in I$, and (2) no failure-free extension of α contains a $decide(1)_i$ 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 $decide(0)_i$ action, for some i, and (2) some failure-free extension of α contains a $decide(1)_i$ action, for some i. These definitions 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 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. Since the execution 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. $\square$

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

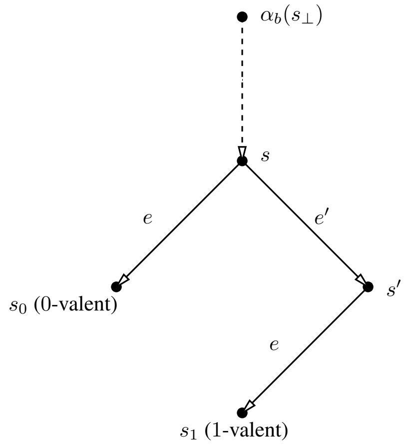

Figure 2: A hook starting in α.

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

Now define an edge-labeled directed graph $G(\mathcal{C})$ as follows:

(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 $\alpha' = e(\alpha)$.

By assumption (i) of Section 4.1, 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, there is at most one edge labeled with e outgoing from α.

4.4 The existence of a hook

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.

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

Proof: 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 2 implies that, for any finite failure-free extension α' of α (such that e is not executed along the suffix of α' starting in the last state of α) e is applicable to α', and hence $e(\alpha')$ is defined. We look for a vertex α' of $G(\mathcal{C})$, reachable from α in $G(\mathcal{C})$ without following any edge labeled with e, 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.

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 is a contradiction, so π must be finite.

```
α ← α_b
while true do
   Let e be the next task (in round-robin order)
         applicable to α
   if α has a descendant α′ in G(C) such that
         the path from α to α′ includes
         no e labels and e(α′) is bivalent then
      choose some such α′
      α ← e(α′)
   else
      exit
```

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

Let α be the last vertex of π. By construction, α is bivalent. Upon termination of the above path construction in vertex α, let e be the next task in round robin order that is applicable to α. Such an e always exists since nonfaulty processes can always take a step, by assumption. Since the path construction terminated in α, we conclude that e satisfies the following condition: for any descendant α' of α, 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 such that $e(\alpha')$ is 1-valent. Let $\sigma_0, \ldots, \sigma_m$ be the sequence of vertices of $G(\mathcal{C})$ on the path from α to α', and 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, $\sigma_{j+1} = e_j(\sigma_j)$. 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 $e' = e_j$. □

4.5 Similarity

In this section, we introduce notions of similarity between system states. These will be used in showing non-existence of a hook, which will yield the contradiction needed for the impossibility proof. First, we define j-similar system states.

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:

(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.

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 all $i \in J$, β contains no output actions of P_i. Assume that in β, no $perform_{i,c}$ or $b_{i,c}$ (i.e., a response) action of any i-* task, $i \in J$, 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-* task of every service and register ($*$ is *perform* or *output*).

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, the termination property for $(f+1)$-resilient consensus implies that there is a finite prefix of $\alpha_0 \cdot \beta$, which we denote by $\alpha_0 \cdot \gamma$, that includes $decide(v)_l$ 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, all dummy actions, and any remaining internal actions of P_i, $i \in J$. Thus, $\alpha_0 \cdot \gamma'$ is a failure-free extension of α_0 that includes $decide(v)_l$. Since α_0 is 0-valent, v must be equal to 0.

We claim that $decide(0)_l$ occurs in the suffix γ', rather than in the prefix α_0. Suppose for contradiction that the $decide(0)_l$ action occurs in the prefix α_0. Then by our technical assumption about processes, the decision value 0 is recorded in the state of 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 $decide(0)_l$ occurs in the suffix γ'.

Now we show how to append essentially the same γ' after α_1. We know that, for every $i \in J$, γ' contains no locally controlled action of P_i, and contains no $perform_{i,c}$ or $b_{i,c}$ action ($b \in resps$), for any $c \in K \cup R$. By definition of j-similarity, we have:

(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 $buffer(i)_c$ is the same in s_0 and s_1.

Thus:

(c) If γ' contains any locally controlled actions of a process i, then the state of P_i is the same in s_0 and s_1.

(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 $perform_{i,c}$ or $b_{i,c}$ ($b \in resps$) actions of c, then the value of $buffer(i)_c$ is the same in s_0 and s_1.

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 $decide(0)_l$.[3] But α_1 is 1-valent — a contradiction. □

Similarly, we define the notion of k-similar states: 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 conditions 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.

[3]Really, we are appending another execution fragment γ'' after α_1 — one that looks the same to all the processes and service tasks that take steps in γ'.

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: Fix k, α_0, α_1, s_0, and s_1 as in the hypotheses of the lemma. By contradiction, 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 all $i \in J$, β contains no output actions of i. Assume that in β, no $perform_{i,k}$ or $b_{i,k}$ action ($b \in resps$) of S_k occurs; we may assume this because the $f+1$ *fail* actions enable *dummy* actions in all 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, the termination property for $f+1$-resilient consensus implies that there is a finite prefix of $\alpha_0 \cdot \beta$, which we denote by $\alpha_0 \cdot \gamma$, that includes $decide(v)_l$ for some $l \in I - J$ and $v \in \{0,1\}$. We know that $decide(0)_l$ occurs in the suffix γ, rather than in the prefix α_0, by an argument similar to that in the proof of Lemma 6.

Now construct $\alpha_0 \cdot \gamma'$, where γ' is obtained from γ by removing all the $fail_i$ actions, $i \in J$, and all *dummy* actions. Thus, $\alpha_0 \cdot \gamma'$ is a failure-free extension of α_0 that includes $decide(v)_l$. Since α_0 is 0-valent, v must be equal to 0.

Now we show how to append essentially the same γ' after α_1. By definition of k-similarity, we have:

(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.

Thus:

(c) For every $c \in K \cup R$, if γ' contains any $perform_{i,c}$ 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 $decide(0)_l$. But α_1 is 1-valent — a contradiction. □

4.6 The non-existence of a hook

Now we are ready to prove the absence of hooks.

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

Proof: By contradiction. Assume that a hook exists, as depicted in Figure 2. 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 (Assumption (i) in Section 4.1), 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 2 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 $\mathit{participants}(e, s) \cap \mathit{participants}(e', s) \neq \emptyset$, either a process, service, 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 $P_i \in \mathit{participants}(e, s) \cap \mathit{participants}(e', s)$. 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 $S_k \in \mathit{participants}(e, s) \cap \mathit{participants}(e', s)$. 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 (but not case 4), whenever $\mathit{action}(e, s)$ and $\mathit{action}(e', s)$ access the same buffer, one action inserts an intem and the other removes an intem. Hence the actions commute.

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

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

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

4. For some $i, j \in I$, $participants(e, s) = \{S_r, P_i\}$ and $participants(e', s) = \{S_r, P_j\}$.
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 $participants(e, s) \cap participants(e', s) = \emptyset$. But this directly contradicts Claim 2. □

Lemma 5 contradicts Lemma 8. Hence we have derived a contradiction by assuming the negation of Theorem 1. Hence Theorem 1 is established.

5 k-Set Consensus

Our boosting impossibility result concerns *consensus* implementations. Interestingly, while it is not possible to implement $(f+1)$-resilient *consensus* using registers and f-resilient atomic objects, this is not the case for the *k-set consensus problem* [6]. In k-set consensus, the processes have to agree on at most k different values (k-set consensus reduces to consensus when $k = 1$).

Consider a set of f-resilient k-set consensus services, each one exporting *m ports*. An algorithm that implements f'-resilient k'-set consensus works as follows. Take a *principal* subset of the processes, and divide it into s disjoint groups, each one accessing a different service. Each principal process participates in an execution proposing its input value to its designated service. When it gets a decision back, the process decides on the value and writes it in a shared register. The remaining processes simply wait until at least one principal process writes the value. The values of k' and f' depend on the size of the principal set, and on the number s of services we divide it into. There is a tradeoff between k' and f': if a small number of failures f' is tolerated, then a high degree of agreement is achieved, namely a small k'. If more failures f' must be tolerated, then a lower degree of agreement is achieved, namely a large k'.

To achieve correctness, we must ensure first that at least one principal process receives a decision from its service and communicates the decision to all, i.e., (1) every f-resilient service is connected to $f+1$ processes, and (2) fewer than $s \cdot (f+1)$ principal processes can fail: $f' < s \cdot (f+1)$. Thus, there is at least one service S that is not killed, and moreover, there is at least one correct principal process that receives a decision value from S and writes the decision in a shared register. Thus, every correct process eventually decides. The number of possible different decision values is at most $s \cdot k$: there are at most k different values returned per service; more precisely, at most k values per service being accessed by at least k processes, and c values for a service that is being accessed by c processes for $c < k$. Thus, for a desired overall resilience f', we want the smallest possible k' and so we find the smallest integer s that guarantees $f' < s \cdot (f+1)$. Thus, we have $s = \lceil (f'+1)/(f+1) \rceil$ services, and take the first $f'+1$ processes to be the principal processes ($f'+1$ processes using as few services as possible, each one with $f+1$ input ports). It follows that

Theorem 9 *For any $1 \leq k < m$, $k \leq f \leq m-1$, $1 \leq f' \leq n-1$, it is possible to implement f'-resilient k'-set consensus using read-write memory and f-resilient k-set consensus services, each one with m ports, for*

$$k' \geq k \cdot \left\lfloor \frac{f'+1}{f+1} \right\rfloor + \min(k,\ (f'+1)\bmod(f+1)).$$

When each available service is wait-free, that is $f = m-1$, this algorithm reduces to the one of [12], and gives a tight bound. As an example, assume that we want to implement a f'-resilient k'-set consensus in a system of $2c$ processes, where $f' = 2c-1$, using only 1-resilient consensus services, i.e., $f = 1$, $k = 1$. The smallest k' for which we can do this is $k' = c$, using $s = c$ services, each shared by 2 processes ($f'+1 = 2c$ principal processes).

Note that the algorithm above uses services that are not connected to all processes. It is known that f-resilient f-set consensus cannot be solved using only reliable registers [2, 13, 19]. We conjecture that f-resilient f-set consensus cannot be solved using only reliable registers and services that are connected to all processes.

6 Failure-Oblivious Services

A *failure-oblivious service* is a generalization of an atomic object. It allows an invocation to trigger multiple processing steps instead of just one *perform* step. These steps can interleave with processing steps triggered by other invocations, and this makes a failure-oblivious service non-atomic, in general. A failure-oblivious service also allows an invocation to trigger any number of responses, at any endpoints, instead of just a single response at the endpoint of the invocation. The service may also include background processing tasks, not related to any specific endpoint. 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 examples, and describe how Theorem 1 can be extended to such services.

6.1 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 for canonical atomic objects. Also, in place of the sequential type parameter $\mathcal{T}$, the service has a *service type parameter* $\mathcal{U}$, which is a tuple $\langle V, V_0, \mathit{invs}, \mathit{resps}, \mathit{glob}, \delta_1, \delta_2, \delta_3\rangle$, where 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 tasks*, and $\delta_1, \delta_2, \delta_3$ are three transition relations.

Here, δ_1 is a total binary relation from $\mathit{invs} \times J \times V$ to (the set of mappings from J to finite sequences of *resps*) $\times V$. It is used to map an invocation at the head of a particular $\mathit{inv}-\mathit{buffer}$, and the current value for *val*, to a set of results, each of which consists of a new value for *val* and sequences of responses to be added to any or all of the $\mathit{resp}-\mathit{buffer}$s. δ_2 is a total binary relation from $J \times V$ to (the set of mappings from J to finite sequences of *resps*) $\times V$. It is used to map a particular endpoint and value of *val* to a set of results, defined as above. Finally, δ_3 is a total binary relation from V to (the set of mappings from J to finite sequences of *resps*) $\times V$. It it used to map a value of *val* to a set of results. The code for a canonical failure-oblivious automaton, showing how these parameters are used, appears in Figure 4.

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

An I/O automaton A is an *f-resilient failure-oblivious service* of type $\mathcal{U}$, endpoint set J, and index k, provided that it implements the canonical f-resilient failure oblivious service S of type $\mathcal{U}$ for J and k, in the same sense as for atomic objects.

6.2 Example: Totally Ordered Broadcast

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 $\mathcal{U} = \langle V, V_0, \mathit{invs}, \mathit{resps}, \mathit{glob}, \delta_1, \delta_2, \delta_3\rangle$. 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 $\{\mathit{bcast}(m) : m \in M\}$. The response set *resps* is $\{\mathit{rcv}(m,i) : m \in M, i \in J\}$. ($\mathit{rcv}(m,i)$ indicates the receipt of message m from sender i. This receipt can occur at any endpoint.) *glob* consists of one task named g, that is, $\mathit{glob} = \{g\}$. δ_1, the relation describing the transitions that process invocations from $\mathit{inv}-\mathit{buffer}$s, is defined in Figure 6:

This code processes the first element of $\mathit{inv}-\mathit{buffer}(i)$ by adding it to the end of the sequence stored in *msgs*. (Formally, $\delta_1((a,i,v),(B,v'))$ holds iff $a = \mathit{bcast}(m)$, $v'.\mathit{msgs}$ is the result of adding (m,i) to the end of $v.\mathit{msgs}$, and $B(j)$ is empty for all j.)

δ_2 is the identity relation, indicating that no other processing is done on behalf of i. Relation δ_3 is defined in Figure 7:

(Formally, $\delta_3(v,(B,v'))$ holds iff either (a) $v.\mathit{msgs}$ is nonempty, $(m,i) = \mathit{head}(v.\mathit{msgs})$, $v'.\mathit{msgs} = \mathit{tail}(v.\mathit{msgs})$, and for every $j \in J$, $B(j)$ is the sequence consisting of the single element $\mathit{rcv}(m,i)$, or (b) $v.\mathit{msgs}$ is empty, $v' = v$, and for every j, $B(j)$ is the empty sequence.)

6.3 Impossibility of Boosting

Let index set K include now the indices of all failure-oblivious services. Now the notion of k-similarity restricts the states of all registers and of all atomic and failure-oblivious services except S_k.

We now argue that Lemmas 2–8 extend to this case.

Lemma 2: We have added the i-compute and g-compute tasks to the definition of a service, Figure 4. These are defined using total transition relations δ_2 and δ_3. Since these are total relations, we see from Figure 4 that these tasks are always enabled. Hence Lemma 2 still holds.

CanonicalFailureObliviousService($\mathcal{U}, J, f, k$),
where $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2, \delta_3 \rangle$

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:
$perform_{i,k}$, $i \in J$
$compute_{i,k}$, $i \in J$
$dummy\_*_{i,k}$, $* \in \{perform, compute, output\}$, $i \in J$
$compute_{g,k}$, $g \in glob$
$dummy\_compute_{g,k}$, $g \in glob$

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

Internal: $compute_{g,k}$, $g \in glob$
Precondition:
$\delta_3(val, (B, v))$
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: $dummy\_*_{i,k}$, $i \in J$
As for canonical atomic object.

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

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

Figure 4: A canonical failure-oblivious service.

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

Figure 5: The composition of *val* in a totally ordered broadcast service.

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

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

Lemmas 3–5: The proofs of these lemmas do not depend on the definition of a service, and so they carry over.

Lemma 6: The proof carries over by replacing every reference to $perform_{i,k}$ actions with a reference to $perform_{i,k}$ or $compute_{i,k}$ or $compute_{g,k}$ actions. We provide a complete proof in Appendix B.

Lemma 7: Since service S_k is "silent" along γ, the change in its definition does not affect the proof. The other services have the same behavior along γ and γ', and the original proof of Lemma 7 does not refer to their detailed definition. Hence this proof carries over.

Lemma 8: Claims 1, 2, 3, and 5 carry over with no difference in the proof, since their proof does not refer to the definition of actions of services. For claim 4, the proof of case 1 ($participants(e, s) = participants(e', s) = \{S_k\}$) must be modified by replacing every reference to $i - perform$ tasks with a reference to $i - perform$ or $i - compute$ or $g - compute$ tasks. The proofs of the other cases carry over. Hence the lemma as a whole carries over. We provide a complete proof in Appendix B.

Hence the following result:

Theorem 10 *Let f and n be integers, $0 \leq f < n-1$. There does not exist an $(f+1)$-resilient n-process implementation of consensus from canonical f-resilient atomic services, canonical f-resilient failure-oblivious services, and canonical reliable registers.*

7 General (Failure-Aware) Services

A *general*, or *failure-aware* service is a further generalization of a failure-oblivious service. This time, the generalization removes the failure-oblivious constraint, allowing the service's decisions to depend on knowledge of failures of processes connected to the service.

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

Figure 7: Relation δ_3 in a totally ordered broadcast service.

7.1 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 failure-oblivious services, and by a service type parameter $\mathcal{U}$, which is a tuple $\langle V, V_0, invs, resps, glob, \delta_1, \delta_2, \delta_3 \rangle$, as for failure-oblivious services. This time, however, the domains of δ_1, δ_2, and δ_3 are $invs \times J \times V \times 2^I$, $J \times V \times 2^I$, and $V \times 2^I$, respectively. The final argument, in each case, will be instantiated in the service code with the current *failed* set.

The only portions of the code that are different from those for failure-oblivious services are the three transition definitions that use the δ_1, δ_2, and δ_3 (Figure 8).

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

Internal: $compute_{g,k}$, $g \in glob$
Precondition:
  $\delta_3((val, failed), (B, v))$
Effect:
  $val \leftarrow v$
  for $j \in J$ do
    add $B(j)$ to end of $resp-buffer(j)$

Figure 8: Relations δ_1, δ_2 and δ_3 in a general service.

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

7.2 Examples: Failure detectors

In this section, we describe how a variety of 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]: because our failure detectors are automata, they cannot predict future input actions. Thus, our services encompass only *realistic* failure detectors [7].

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

7.2.1 Perfect Failure Detector $\mathcal{P}$

First, we define an f-resilient perfect failure detector for J and k. V contains only one (trivial) state, that is, the service maintains no internal information other than the *failed* set. Responses are of the form $suspect(J')$, $J' \subseteq J$. The set *glob* of global tasks is empty. Since there are no invocations, δ_1 is trivial. Since there are no global tasks, δ_3 is empty. All that remains is to define δ_2, which describes computation on behalf of each process i: $\delta_2(i, failed)$ simply puts a *suspect* response containing the current *failed* set into i's response buffer (Figure 9).

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

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

7.2.2 Eventually Perfect Failure Detector $\diamond\mathcal{P}$

Again, responses are of the form $suspect(J')$, $J' \subseteq J$. We model eventual perfection using a *mode* variable, which can take on values *perfect* or *imperfect*. Initially, and after each new failure, *mode* is set to *imperfect*. A background task is responsible for eventually switching *mode* to *perfect*. Since failures must eventually stop, the *mode* eventually remains *perfect*. While in *perfect* mode, the failure detector suspects exactly the processes that have failed. In *imperfect* mode, suspicions are arbitrary. The set of internal state components in $\diamond\mathcal{P}$ is presented in Figure 10.

Components of val:
  $mode \in \{perfect, imperfect\}$, initially $imperfect$
  $oldfailed \subseteq J$, initially $\emptyset$

Figure 10: The composition of *val* in $\diamond\mathcal{P}$.

The global task set $glob = \{g_1, g_2\}$. Task g_1 is responsible for setting *mode* to *imperfect* while task g_2 sets it to *perfect*. The interesting transition definitions are presented in Figure 11.

Internal: $compute_{i,k}$
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: $compute_{g_1,k}$
Precondition:
  true
Effect:
  if $failed \neq oldfailed$ then
    $mode := imperfect$
    $oldfailed := failed$

Internal: $compute_{g_2,k}$
Precondition:
  true
Effect:
  if $failed = oldfailed$ then
    $mode := perfect$

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

7.2.3 Eventual Leader Service Ω

The eventual leader service Ω provides $leader(l)$ responses at all nodes, where $l \in J$. Eventually (assuming that not all processes fail), the latest *leader* announcements should be identical at all endpoints, and should indicate the name of a non-failed endpoint. We again model eventual perfection using a *mode* variable (Figure 12).

Components of val:
  $mode \in \{perfect, imperfect\}$, initially $imperfect$
  $oldfailed \subseteq J$, initially $\emptyset$
  $leader \in J \cup \{\perp\}$, initially $\perp$

Figure 12: The composition of val in Ω.

We again use two global tasks g_1, g_2. Now g_1 sets *mode* to *imperfect* and removes any choice of leader, while g_2 sets *mode* to *perfect* and chooses a leader. The corresponding transition definitions are presented in Figure 13.

7.3 Impossibility of Boosting

Our impossibility results for atomic and failure-oblivious services allow arbitrary connections between processes and services. However, it turns out that we

Internal: $compute_{i,k}$
Precondition:
  true
Effect:
  if $mode = perfect$ then
    add $leader(leader)$ to $resp-buffer(i)$
  else
    choose $j \in J$
    add $leader(j)$ to $resp-buffer(i)$

Internal: $compute_{g_1,k}$
Precondition:
  true
Effect:
  if $failed \neq oldfailed$ then
    $leader := \perp$
    $mode := imperfect$
    $oldfailed := failed$

Internal: $compute_{g_2,k}$
Precondition:
  true
Effect:
  if $failed = oldfailed \wedge leader \neq \perp$ then
    $leader :=$ choose l where $l \in J - failed$
    $mode := perfect$

Figure 13: Internal transitions in Ω.

can boost the resilience of systems containing failure-aware services, if we allow arbitrary connection patterns:

For example, consider a system that uses wait-free registers and 1-resilient perfect failure detectors. Suppose that every pair of processes shares a 1-resilient 2-process failure detector. Such a system can implement a *wait-free* perfect failure detector for all 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 outputs its set of suspected processes. Since every perfect failure detector is 1-resilient, the algorithm is wait-free. Using this construction, f-resilient consensus, for any f, can be implemented using wait-free registers and 1-resilient services.

This boosting is, however, impossible if we assume a system in which f-resilient failure-aware services must be connected to all processes, thus, $f + 1$ process failures overall can disable all the failure-aware services. We assume that the system may also contain f-resilient failure-oblivious services, connected to arbitrary processes. By applying arguments similar to ones presented in Section 4, we can prove boosting to be impossible, i.e., that $(f + 1)$-resilient consensus cannot be solved in such a model.

The proof is also based on analysis of a "hook". In fact, we need to introduce only slight modifications into the proofs of Lemmas 6 and 7: Let α_0 and α_1 be any two univalent failure-free input-first executions whose respective final states, s_0 and s_1, are j-similar (respectively, k-similar). Assume, by contradiction, that α_0 and α_1 have opposite valences. The definitions of j-similarity and k-similarity do not restrict the states of failure-aware services, that is, failure-aware services can have arbitrary states in s_0 and s_1, the respective final states of α_0 and α_1.

However, note that the $f+1$ failures of processes in J allow every failure-aware service to stop performing (non-dummy) locally controlled steps. Then following the arguments of Lemmas 6 and 7, we can construct a failure-free extension of α_0, $\alpha_0 \cdot \gamma'$, such that (1) γ' includes $decide(v)_l$, for some $l \in I - J$; (2) γ' includes no locally controlled step of process P_j, nor any $perform_j$, $compute_j$, or $output_j$ step for any service or register (respectively, γ' includes no locally controlled step of service S_k); (3) γ' includes no locally controlled step of any failure-aware service. Thus, γ' is essentially applicable to α_1 — a contradiction with the assumption that α_0 and α_1 have opposite valences.

We first note that Lemmas 2–5 carry over to the case of general services. The argument for this is identical to that for failure-oblivious services, given in Section 6.3.

For Lemma 6: The proof for the case of failure oblivious services already handles both atomic and failure oblivious services. To handle f-resilient general services, we note that we can assume that all of these servies are "silent" along γ, since the occurrence of $f+1$ $fail_i$ actions enables a dummy action in every task of every general service. Thus the different definition for actions $perform_{i,k}$, $compute_{i,k}$ and $compute_{g,k}$, in particular, their ability to observe the set of failed processes, makes no difference. Hence γ' can be appended after α_1 in the same way as in the proof for the case of failure oblivious services.

For Lemma 7: Since the service S_k can be "silenced" as before, the proof is unchanged from that for failure oblivious services.

For Lemma 8: We defined the hook so that it does not contain any $fail_i$ actions. Hence at all states in the hook, the set *failed* of failed processes is empty. Thus the different definition for actions $perform_{i,k}$, $compute_{i,k}$ and $compute_{g,k}$, 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.

Hence the following result:

Theorem 11 *Let f and n be integers, $0 \le f < n-1$. There does not exist an $(f+1)$-resilient n-process implementation of consensus from canonical f-resilient general services connected to all processes, canonical f-resilient atomic services (connected to arbitrary processes), canonical f-resilient failure-oblivious services (connected to arbitrary processes), and canonical reliable registers.*

8 Conclusions

We have established the impossibility of boosting the resilience of services in a distributed asynchronous system where processes are subject to undetectable stopping failures. Our results can be viewed as a generalization to any number f of failures of the impossibility result of Fischer, Lynch and Paterson [8] for $f = 1$. While our first result (for atomic objects) can be derived from existing results in the literature, the direct proof that we give is simpler, and is also easily extended to more general services than atomic objects.

References

[1] P. 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.

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

[3] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94)*, pages 334–343, August 1994.

[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. On failure detectors and type boosters. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, October 2003.

[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 S. Rajsbaum. Algebraic spans. *Mathematical Structures in Computer Science (Special Issue: Geometry and Concurrency)*, 10(4):549–573, August 2000.

[13] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, November 1999.

[14] 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.

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

[16] 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.

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

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

[19] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29:1449–1483, 2000.

Appendix A Alternative proof for atomic services

In this section, we show how our result for the case of atomic objects can be derived from earlier results [3, 11, 16, 17]. This alternative proof of our result was obtained independently and concurrently by Jayanti [15] and Guerraoui and Kouznetsov [9]. However, this alternative proof does not extend to more general services.

A.1 The proof

The following two lemmas are restatements in our terminology of the "necessity" part and the "sufficiency" part of Theorem 4.1 in [3], respectively.

Lemma 12 *Let f and n be integers, $0 \leq f$, $1 \leq n$. Then there exists an f-resilient n-process implementation of consensus from wait-free $(f+1)$-process consensus objects and reliable registers.*[4]

Lemma 13 *Let f and n be integers, $2 \leq f < n$. Then there exists a wait-free $(f+1)$-process implementation of consensus from f-resilient n-process consensus objects and reliable registers.*

The following result follows easily from Herlihy's universal construction [11]:

Lemma 14 *Let f and n be integers, $0 \leq f$, $1 \leq n$. Let $\mathcal{T}$ be a sequential type. Then there exists an f-resilient n-process implementation of an atomic object of type $\mathcal{T}$ from f-resilient n-process consensus objects and reliable registers.*

The following result is shown in [16].

Lemma 15 *Let n be integer, $n \geq 0$. There does not exist a wait-free $(n+1)$-process implementation of consensus from wait-free n-process consensus objects and reliable registers.*

Theorem 1 *Let f and n be integers, $0 \leq f < n-1$. There does not exist an $(f+1)$-resilient n-process implementation of consensus from f-resilient atomic objects and reliable registers.*

Proof: By contradiction, assume that there exists an $(f+1)$-resilient n-process implementation of consensus from f-resilient atomic objects and reliable registers. We consider two cases.

First suppose that $f = 0$, so $n \geq 2$. Thus, we have a 1-resilient n-process implementation of consensus using 0-resilient atomic objects and reliable registers. By Lemma 14, each 0-resilient atomic object used in this implementation can itself be implemented from 0-resilient consensus objects and reliable registers. By substituting these implementations for the objects, we obtain a 1-resilient n-process implementation of consensus using 0-resilient consensus objects and reliable registers. Now, a 0-resilient consensus object can be implemented from reliable registers,[5] so substituting

[4] Theorem 4.1 in [3] assumes $2 \leq f$. However, the necessity part of the theorem requires only $0 \leq f$.

[5] A 0-resilient consensus with an endpoint set J can be easily implemented from two reliable registers as follows. Every process participating in the consensus algorithm writes its input value in a dedicated "proposal" register R (initialized to $\perp$). Then the process keeps reading a dedicated "decision" register D (initialized to $\perp$) until a non-$\perp$ value is read, in which case the process decides on this value. In parallel, a dedicated process P_i ($i \in J$) keeps reading R. As soon as P_i reads a non-$\perp$ value v in R, P_i writes v in D.

once more, we obtain a 1-resilient n-process implementation of consensus using only reliable registers. But this contradicts the impossibility result of [17].

Now suppose that $f \geq 1$. By Lemma 14, each f-resilient atomic object used in this implementation can itself be implemented from f-resilient consensus objects and reliable registers. By substituting, we obtain an $(f+1)$-resilient n-process implementation of consensus from f-resilient consensus objects and reliable registers. By Lemma 12, each f-resilient consensus object used in this implementation can be implemented from wait-free $(f+1)$-process consensus objects and reliable registers. By substituting again, we obtain an $(f+1)$-resilient n-process implementation of consensus from wait-free $(f+1)$-process consensus objects and reliable registers. Now by Lemma 13 (using the fact that $2 \leq f+1 < n$), a wait-free $(f+2)$-process consensus object can be implemented from $(f+1)$-resilient n-process consensus objects and reliable registers. By substituting, we obtain an implementation of a wait-free $(f+2)$-process consensus object from wait-free $(f+1)$-process consensus objects and reliable registers. But this contradicts Lemma 15. □

A.2 Extension to more general services

The argument in the previous subsection does not extend to *all* services. Here we give two reasons for this.

First, the universality result fails to hold for many distributed services. In particular, no meaningful failure detector can be implemented from consensus objects. Indeed, by definition, an atomic service does not provide any information about failures: the value of the service is not affected by failures of processes. Here we simply give an example, showing that consensus cannot implement a perfect failure detector.

Indeed, assume, by contradiction, that there is an algorithm A that implements a perfect failure detector in a system of n processes using n-process consensus objects and registers. Consider any finite execution α of A in which process i is faulty and is declared to be faulty. Now we consider an execution α' that is identical to α except that α' includes no $fail_i$ event (i is just slow to take steps in α'). Clearly, α' is also a finite execution of A, since registers and consensus objects are failure-oblivious. Thus, in α', a process is declared faulty without having failed— a contradiction.

The second reason why the arguments of [3] do not work with non-atomic services is that, generally speaking, an f-resilient implementation of n-process consensus is not equivalent to a wait-free implementation of $(f+1)$-process consensus (Theorem 4.1 of [3]). Indeed, if f-resilient k-process consensus is implemented from non-atomic services, the simulation algorithm presented in the proof of Theorem 4.1 in [3] is not valid: a step of a process accessing a general service cannot always be simulated by another process. This is because a response of a non-atomic service to a given process i might not necessarily be simulated by another process j without communicating with i, i.e., no set of $f+1$ processes can independently simulate an f-resilient k-process consensus algorithm without communicating with the rest of the system.

Appendix B Complete proofs for failure-oblivious services

Proof of Lemma 6 when failure-oblivious services are allowed.

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. Without loss of generality, assume that all services are failure-oblivious. Atomic services can be handled by the same argument as used in the proof of Lemma 6 for atomic services only.

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 all $i \in J$, β contains no output actions of P_i. Assume that in β, no $perform_{i,c}$, $compute_{i,c}$, or $b_{i,c}$ action of any i-* task, $i \in J$, 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 task of every service and register ($*$ is *perform* or *compute* or *output*).

Further assume that in β, no $compute_{g,c}$ action of any g-*compute* task occurs at any component $c \in K \cup R$; we may assume this because the occurrence of $f+1$ $fail_i$ actions enables the $dummy-compute_{g,c}$ action in every g-*compute* task of every failure-oblivious service c.

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, the termination property for $(f+1)$-resilient consensus implies that there is a finite prefix of $\alpha_0 \cdot \beta$, which we denote by $\alpha_0 \cdot \gamma$, that includes $decide(v)_l$ 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, all dummy actions, and any remaining internal actions of P_i, $i \in J$. Thus, $\alpha_0 \cdot \gamma'$ is a failure-free extension of α_0 that includes $decide(v)_l$. Since α_0 is 0-valent, v must be equal to 0.

We claim that $decide(0)_l$ occurs in the suffix γ', rather than in the prefix α_0. Suppose for contradiction that the $decide(0)_l$ action occurs in the prefix α_0. Then by our technical assumption about processes, the decision value 0 is recorded in the state of 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 $decide(0)_l$ occurs in the suffix γ'.

Now we show how to append essentially the same γ' after α_1. We know that, for every $i \in J$, γ' contains no locally controlled action of P_i, and contains no $perform_{i,c}$, $compute_{i,c}$, or $b_{i,c}$ action, for any $c \in K \cup R$. By definition of j-similarity and $j \in J$, we have:

(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 $buffer(i)_c$ is the same in s_0 and s_1.

Thus:

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

(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 $perform_{i,c}$, $compute_{i,c}$, or $output_{i,c}$ actions, then $i \notin J$, and so the value of $buffer(i)_c$ is the same in s_0 and s_1.

Finally, we note that the presence of $compute_{g,c}$ does not invalidate the argument. A $compute_{g,c}$ cannot refer to or modify any input buffers. The precondition of $compute_{g,c}$ depends only on val_c, and so the same $compute_{g,c}$ actions can be applied in γ' after α_1, and they can add the same items to the output buffers. Thus for $i \notin J$ the sequence of values that $buffer(i)_c$ takes along γ' after α_0 and γ' after α_1 are the same.

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 $decide(0)_l$.[6]

But α_1 is 1-valent — a contradiction. □

Proof of Lemma 7 when failure-oblivious services are allowed.

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: Fix k, α_0, α_1, s_0, and s_1 as in the hypotheses of the lemma. By contradiction, 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 all $i \in J$, β contains no output actions of i. Assume that in β, no $perform_{i,k}$ or $b_{i,k}$ or $compute_{i,k}$ or $compute_{g,k}$ action ($b \in resps, g \in glob$) of S_k occurs; we may assume this because the $f+1$ $fail$ actions enable $dummy$ actions in all 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, the termination property for $f+1$-resilient consensus implies that there is a finite prefix of $\alpha_0 \cdot \beta$, which we denote by $\alpha_0 \cdot \gamma$, that includes $decide(v)_l$ for some $l \in I - J$ and $v \in \{0,1\}$. We know that $decide(0)_l$ occurs in the suffix γ, rather than in the prefix α_0, by an argument similar to that in the proof of Lemma 6.

Now construct $\alpha_0 \cdot \gamma'$, where γ' is obtained from γ by removing all the $fail_i$ actions, $i \in J$, and all $dummy$ actions. Thus, $\alpha_0 \cdot \gamma'$ is a failure-free extension of α_0 that includes $decide(v)_l$. Since α_0 is 0-valent, v must be equal to 0.

Now we show how to append essentially the same γ' after α_1. By definition of k-similarity, we have:

(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.

Thus:

[6] Really, we are appending another execution fragment γ'' after α_1 — one that looks the same to all the processes and service tasks that take steps in γ'.

(c) For every $c \in K \cup R$, if γ' contains any $perform_{i,c}$ or $b_{i,c}$ or $compute_{i,k}$ or $compute_{g,k}$ 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 $decide(0)_l$. But α_1 is 1-valent — a contradiction. □

Proof of Lemma 8 when failure-oblivious services are allowed.

Lemma 8 We establish the same 5 claims as in the case of atomic services, which establishes the needed contradiction.

Claims 1, 2, and 5 do not refer to the definition of a service, and so their proof remains unchanged from the atomic services case.

The proof of Claim 3 is unchanged, since the only actions considered have as participants either a process P_i, or P_i and a component S_c, $c \in K \cup R$. Thus, whenever S_c is a participant, the action must be an external action of S_c.

Since the external actions in the definitions of atomic service and failure oblivious service have the same effect, namely to add or remiove a single item from a single buffer, it follows that the proof of Claim 3 for the atomic case still applies.

The proof of Claim 4 is modified as follows.

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

1. $participants(e, s) = participants(e', s) = \{S_k\}$.

   Then e and e' must be $i-perform$ or $i-compute$ 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 $i \in I$, $participants(e, s) = \{S_k, P_i\}$ and $participants(e', s) = \{S_k\}$.

   Hence $action(e, s)$ is either $a_{i,k}$ or $b_{i,k}$, and $action(e', s)$ is one of $perform_{j,k}$, $compute_{j,k}$, or $compute_{g,k}$, where $j \in J_k, g \in glob$.

   Inspection of the definition of a failure-oblivious service shows that the two tasks commute, that is, $e'(s_0) = s_1$ — a contradiction.

3. For some $i \in I$, $participants(e', s) = \{S_k, P_i\}$ and $participants(e, s) = \{S_k\}$.

   Hence $action(e, s)$ is one of $perform_{j,k}$, $compute_{j,k}$, or $compute_{g,k}$, where $j \in J_k, g \in glob$, and $action(e', s)$ is either $a_{i,k}$ or $b_{i,k}$.

   Inspection of the definition of a failure-oblivious service shows that the two tasks commute, that is, $e'(s_0) = s_1$ — a contradiction.

4. For some $i, j \in I$, $participants(e, s) = \{S_k, P_i\}$ and $participants(e', s) = \{S_k, P_j\}$.

   By Claim 3, we know that $i \neq j$. Now $action(e, s)$ is either $a_{i,k}$ or $b_{i,k}$, and $action(e', s)$ is either $a_{j,k}$ or $b_{j,k}$.

   Inspection of the definition of a failure-oblivious service shows that the two tasks commute, that is, $e'(s_0) = s_1$ — a contradiction.