Impossibility of boosting distributed service resilience

Paul Attie, Rachid Guerraoui, Petr Kouznetsov, Nancy Lynch, and Sergio Rajsbaum
The Impossibility of Boosting Distributed Service Resilience *

Paul Attie\textsuperscript{1,3} Rachid Guerraoui\textsuperscript{2} Petr Kouznetsov\textsuperscript{2} Nancy Lynch\textsuperscript{3} Sergio Rajbsbaum\textsuperscript{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-reader multi-writer 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].

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

\*\*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 \( \alpha \) of \( A \) is fair iff for each task \( e \) of \( A \): (1) if \( e \) is finite, then \( e \) is not enabled in the final state of \( \alpha \), and (2) if \( \alpha \) is infinite, then \( \alpha \) contains either infinitely many actions of \( e \), or infinitely many
occurrences of states in which \( \epsilon \) 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 \( \alpha \) and \( \alpha' \) are execution fragments of \( A \) (with \( \alpha \) finite) such that \( \alpha' \) starts in the last state of \( \alpha \), then the concatenation \( \alpha \cdot \alpha' \) is defined, and is called an extension of \( \alpha \).

### 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 \( T = (V, V_0, \text{init}, \text{resps}, \delta) \) consists of:

- \( V \), a nonempty set of values,
- \( V_0 \subseteq V \), a nonempty set of initial values,
- \( \text{init} \), a set of invocations,
- \( \text{resps} \), a set of responses, and
- \( \delta \), a binary relation from \( \text{init} \times V \) to \( \text{resps} \times V \) that is total, in the sense that, for every \( (a, v) \in \text{init} \times V \), there is at least one \( (b, v') \in \text{resps} \times V \) such that \( ((a, v), (b, v')) \in \delta \).

We sometimes use dot notation, writing \( T.V, T.V_0, T.\text{init} \ldots \) for the components of \( T \). We say that \( T \) is deterministic if \( V_0 \) is a singleton set \( \{v_0\} \), and \( \delta \) is a mapping, that is, for every \( (a, v) \in \text{init} \times V \), there is exactly one \( (b, v') \in \text{resps} \times V \) 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”, \( V_0 = \{v_0\} \), \( \text{init} = \{\text{read} \} \cup \{\text{write}(v) : v \in V\} \), \( \text{resps} = V \cup \{\text{ack}\} \), and \( \delta = \{(\text{read}, v, (v, v)) : v \in V \} \cup \{(\text{write}(v), (v', v)) : v \in V \} \cup \{(\text{ack}, v, (v, v')) : (a, v), (b, v') \in \delta \} \).

**Example. Binary consensus sequential type:** Here, \( V = \{\emptyset, \{1\}, \{0\}\}, V_0 = \emptyset \), \( \text{init} = \{\text{init}(v) : v \in \{0, 1\}\} \), \( \text{resps} = \{\text{decide}(v) : v \in \{0, 1\}\} \), and \( \delta = \{(\text{init}(v), \emptyset, (\text{decide}(v), (v, v))) : v \in V \} \cup \{(\text{init}(v), \{1\}, (\text{decide}(v'), (v', v'))) : v, v' \in V \} \).

**Example. \( k \)-consensus sequential type:** Now \( V \) is the set of subsets of \( \{0, 1, \ldots, k\} \) having at most \( k \) elements, \( V_0 = \emptyset \), \( \text{init} = \{\text{init}(v) : v \in \{0, 1, \ldots, k\}\} \), \( \text{resps} = \{\text{decide}(v) : v \in \{0, 1, \ldots, k\}\} \), and \( \delta = \{(\text{init}(v), W), (\text{decide}(v'), W \cup \{v\}) : |W| < k, v', \in W \cup \{v\} \} \cup \{(\text{init}(v), W), (\text{decide}(v'), W) : |W| = k, v', \in W \} \).

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 \( T \) for endpoint set \( J \) and index \( k \), where

- \( T \) is a sequential type,
- \( J \) is a finite set of endpoints at which invocations and responses may occur,
- \( f \in \mathbb{N} \) 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 \( T.\delta \) applied to the invoked and the current value of \( \text{init} \).

We model a failure at an endpoint \( i \) by an explicit input action \( \text{fail}_i \). 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 \( \text{perform},_i \) 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 \( k \)-task \( (\ast \text{ perform or output}) \) contains a special \( \text{dummy},_{i,k} \) action, which is enabled when either process \( i \) has failed or more than \( f \) processes in \( J \) have failed. The \( \text{dummy},_{i,k} \) action is intended to allow, but not force, the service to stop performing
CanonicalAtomicObject\((T, J, f, k)\),
where \(T = (V, \mathcal{V}_0, \text{invs, resp}, \delta)\)

**Signature:**

**Inputs:**
\(a_{i,k}, a \in \text{invs}, i \in J, \text{the invocations at endpoint } i\)
\(\text{fail}_i, i \in J\)

**Outputs:**
\(b_{i,k}, b \in \text{resp}, i \in J, \text{the responses at endpoint } i\)

**Internals:**
\(\text{perform}_{i,k}, i \in J\)
\(\text{dummy}_{i,k}, i \in J\)
\(\{\text{perform, output}\}, i \in J\)

**State components:**
\(\text{val} \in V, \text{initially an element of } \mathcal{V}_0\)
\(\text{inw} \rightarrow \text{buffer}, \text{a mapping from } J \text{ to finite sequences of invs, initially identicaly empty}\)
\(\text{resp} \rightarrow \text{buffer}, \text{a mapping from } J \text{ to finite sequences of resp, initially identicaly empty}\)
\(\text{failed} \subseteq J, \text{initially } \emptyset\)

**Transitions:**

**Input:**
\(a_{i,k}\)

**Effect:**
add \(a\) to end of \(\text{inw} \rightarrow \text{buffer}(i)\)

**Internal:**
\(\text{perform}_{i,k}\)

**Precondition:**
\(\alpha = \text{head}(\text{inw} \rightarrow \text{buffer}(i))\)
\(\delta((\alpha, \text{val}_i, (b_{i,k})))\)

**Effect:**
\(\text{remove head of } \text{inw} \rightarrow \text{buffer}(i)\)
\(\text{val} \leftarrow v\)
\(\text{add } b \text{ to end of } \text{resp} \rightarrow \text{buffer}(i)\)

**Output:**
\(b_{i,k}\)

**Precondition:**
\(b = \text{head}(\text{resp} \rightarrow \text{buffer}(i))\)

**Effect:**
\(\text{remove head of } \text{resp} \rightarrow \text{buffer}(i)\)

**Input:**
\(\text{fail}_i\)

**Effect:**
\(\text{failed} \leftarrow \text{failed} \cup \{i\}\)

**Internal:**
\(\text{dummy}_{i,k}\)

**Precondition:**
\(i \in \text{failed} \lor \text{failed} > f \lor \text{failed} = J\)

**Effect:**
\(\text{none}\)

**Tasks:**
For every \(i \in J:\)
\(\text{perform} = \{\text{perform}_{i,k}, \text{dummy}_{i,k}, \text{dummy}_{i,k}\}\)
\(\text{output} = \{b_{i,k} \in \text{resp}\} \cup \{\text{dummy}_{i,k}\}\)

---

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 \(\text{dummy}_{i,k}\) 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 \(\text{dummy}_{i,k}\) 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 \textit{read/write} is called a \textit{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 \(T\) for endpoint set \(J\) and index \(k\), provided that it implements the canonical \(f\)-resilient atomic object \(S\) of type \(T\) for \(J\) and \(k\), in the following sense:

1. \(A\) and \(S\) have the same input actions (including \textit{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 \textit{wait-free} (or, \textit{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 \(|J|\)-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 \textit{services}). For this section, we fix \(I, K, R, T\) finite (disjoint) index sets for processes, services, and registers, respectively, and \(T\), a sequential type, representing the problem the system is intended to solve. A \textit{distributed system} for \(I, K, R, T\) is the composition of the following I/O automata (see [18, chapter 8]):

1. \textit{Processes} \(P, i \in I\).
2. \textit{Services (atomic objects)} \(S, k \in K\). We let \(T\) 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 $a_i$, $a \in T.i$ and outputs $b_i, b \in T.i$.
  These represent $P_i$'s interactions with the external world.

- For every service $S_k$ such that $i \in J_k$, outputs $a_{i,k}$, $a \in T.k$ and inputs $b_{i,k}, b \in T.k$.

- 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 receive 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 actions 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 $T_k$ for $J_k$ and $k$. Likewise, we assume that register $S_r$ is the canonicalwait-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 $C$ 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 $C$, 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$_{i,k}$, perform$_{i,c}$\}, and $i$-output, consisting of \{b$_{i,k}$ : $b \in T_k$ and $resps$\} $\cup$ \{dummy$_{i, 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 $a$ iff some action of $e$ is enabled in the last state of $a$.

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 \( inv() \) 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 \( C \) 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, \( C \) 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 \( \delta \) 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 \( \alpha \), an applicable task \( e \) determines a unique transition, arising from running task \( e \) from the final state \( s \) of \( \alpha \). 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 \( \alpha \), then \( transition(e, s), first(e, s), action(e, s), \) and \( last(e, s) \) are defined if \( e \) is applicable to \( \alpha \).

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 \( C \). 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 \( \alpha \) be any finite failure-free execution of \( C \), \( e \) be any task of \( C \) applicable to \( \alpha \), and \( \alpha \cdot \beta \) be any failure-free extension of \( \alpha \) such that \( \beta \) 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 \( \alpha \) means that service \( S_k \) has either a pending invocation in an \( inv-buffer \) or a pending response in a \( resp-buffer \) after \( \alpha \). Since \( \beta \) 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 \( C \) arising after a finite failure-free execution \( \alpha \) of \( C \), and let \( e \) be a task that is applicable to \( \alpha \) (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 \( \{P_i, S_k\} \), 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 \( C \) to be a finite execution of \( C \) containing exactly one \( \text{init}(i) \) action for each \( i \in I \), and no other actions. An execution \( \alpha \) of \( C \) is \text{input-first} if it has an initialization as a prefix, and contains no other \( \text{init}(i) \) actions. A finite failure-free input-first execution \( \alpha \) is defined to be \text{0-valent} if (1) some failure-free extension of \( \alpha \) contains a \( \text{decide}(0)_i \) action, for some \( i \in I \), and (2) no failure-free extension of \( \alpha \) contains a \( \text{decide}(1)_i \) action, for any \( i \in I \). The definition of a \text{1-valent} execution is symmetric. A finite failure-free input-first execution \( \alpha \) is \text{univalent} if it is either 0-valent or 1-valent. A finite failure-free input-first execution \( \alpha \) is \text{bivalent} if (1) some failure-free extension of \( \alpha \) contains a \( \text{decide}(0)_i \) action, for some \( i \), and (2) some failure-free extension of \( \alpha \) contains a \( \text{decide}(1)_i \) action, for some \( i \). These definitions immediately imply the following result:

\textbf{Lemma 3} Every finite failure-free input-first execution of \( C \) is either bivalent or univalent.

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

\textbf{Lemma 4} \( C \) has a bivalent initialization.

\textbf{Proof:} Write \( I = \{1, \ldots, n\} \). For each \( i \in \{0, \ldots, n\} \), let \( \alpha^i \) be an initialization of \( C \) 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 \( C \) and Lemma 3, \( \alpha^0 \) is 0-valent, \( \alpha^n \) is 1-valent, and every \( \alpha^j (j \in \{0, \ldots, n\}) \) is either univalent or bivalent.

Then there must be some index \( i \in \{0, \ldots, n-1\} \) such that \( \alpha^i \) is 0-valent and \( \alpha^{i+1} \) is either 1-valent or bivalent. The only difference between the initializations in \( \alpha^i \) and \( \alpha^{i+1} \) is the initial value of \( P_i \). So consider a failure-free extension of \( \alpha^i \) 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 \( \alpha^i \) is 0-valent, the decision must be 0.

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

For the rest of this section, fix \( \alpha_0 \) to be any particular bivalent initialization of \( C \).

\textbf{Figure 2:} A hook starting at \( \alpha_0 \).

4.3 The graph \( G(C) \)

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

(1) The vertices of \( G(C) \) are the finite failure-free input-first extensions of the bivalent initialization \( \alpha_0 \).

(2) \( G(C) \) contains an edge labeled with task \( e \) from \( \alpha \) to \( \alpha' \) provided that \( \alpha' = e(\alpha) \).

By assumption (i) of Section 4.1, any task triggers at most one transition after a failure-free execution \( \alpha \). Therefore, for any vertex \( \alpha \) of \( G(C) \) and any task \( e \), there is at most one edge labeled with \( e \) outgoing from \( \alpha \).

4.4 The existence of a hook

We show that decisions in \( C \) 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 \( G(C) \) of the form depicted in Figure 2.

\textbf{Lemma 5} \( G(C) \) contains a hook.

\textbf{Proof:} Starting from the bivalent vertex \( \alpha_0 \) of \( G(C) \), we generate a path \( \pi \) in \( G(C) \) that passes through bivalent vertices only, as follows. We consider all tasks in a round-robin fashion. Suppose we have reached a bivalent execution \( \alpha \) so far, and task \( e \) is the next task in the round-robin list that is applicable to \( \alpha \). (We know such a task exists because the process tasks are always applicable.)

Lemma 2 implies that, for any finite failure-free extension \( \alpha' \) of \( \alpha \) (such that \( e \) is not executed along the
suffix of $\alpha'$ starting in the last state of $\alpha$) $e$ is applicable to $\alpha'$, and hence $e(\alpha')$ is defined. We look for a vertex $\alpha'$ of $G(C)$, reachable from $\alpha$ in $G(C)$ without following any edge labeled with $e$, such that $e(\alpha')$ is bivalent. If no such vertex $\alpha'$ 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 $\pi$ be the path generated by this construction.

First suppose that $\pi$ is infinite. Then $\pi$ corresponds to a fair failures-free input-first execution $\alpha$ of $C$. Moreover, every finite input-first prefix of $\alpha$ is bivalent. Thus, no process can decide in $\alpha$ (for otherwise, the agreement property of $C$ would be violated). This is a contradiction, so $\pi$ must be finite.

1: $\alpha \leftarrow \alpha_0$
2: while true do
3:   Let $e$ be the next task (in round-robin order) applicable to $\alpha$
4:   if $\alpha$ has a descendant $\alpha'$ in $G(C)$ such that
5:     the path from $\alpha$ to $\alpha'$ includes
6:     no $e$ labels and $e(\alpha')$ is bivalent then
7:     choose some such $\alpha'$
8:     $\alpha \leftarrow e(\alpha')$
9: else
10:   exit

Figure 3: Hook location in $G(C)$.

Let $\alpha$ be the last vertex of $\pi$. By construction, $\alpha$ is bivalent. Upon termination of the above path construction in vertex $\alpha$, let $e$ be the next task in round robin order that is applicable to $\alpha$. Such an $e$ always exists since nonfaulty processes can always take a step, by assumption. Since the path construction terminated in $\alpha$, we conclude that $e$ satisfies the following condition: for any descendant $\alpha'$ of $\alpha$, such that the path from $\alpha$ to $\alpha'$ includes no $e$ labels, $e(\alpha')$ is univalent.

Without loss of generality, assume that $e(\alpha)$ is 0-valent. Since $\alpha$ is bivalent, there is a descendant $\alpha'$ of $\alpha$ such that $e(\alpha')$ is 1-valent. Let $\sigma_0, \ldots, \sigma_m$ be the sequence of vertices of $G(C)$ on the path from $\alpha$ to $\alpha'$, and for each $j$, $0 \leq j \leq m - 1$, let $e_j$ be the label of the edge on this path from $\sigma_j$ to $\sigma_{j+1}$. 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 $\alpha' = \sigma_j$. \qed

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 $C$. Then $s_0$ and $s_1$ are $j$-similar if:

1. For every $i \in I \setminus \{j\}$, the state of $P_i$ is the same in $s_0$ and $s_1$.
2. For every $e \in K \cup R$:
   1. The value of $val_e$ is the same in $s_0$ and $s_1$.
   2. For every $i \in J_c \setminus \{j\}$, the value of $buffer(i)_e$ is the same in $s_0$ and $s_1$.

Lemma 6 Let $j \in I$. Let $\alpha_0$ and $\alpha_1$ be finite failure-free input-first executions, $s_0$ and $s_1$ the respective final states of $\alpha_0$ and $\alpha_1$. Suppose that $s_0$ and $s_1$ are j-similar. If $\alpha_0$ and $\alpha_1$ are univalent, then they have the same valence.

Proof: We proceed by contradiction. Fix $j$, $\alpha_0$, $\alpha_1$, $s_0$, and $s_1$ as in the hypotheses of the lemma, and suppose (without loss of generality) that $\alpha_0$ is 0-valent and $\alpha_1$ 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 $I \setminus J$ is nonempty.

Consider a fair extension of $\alpha_0, \alpha_1 \cdot \beta$, in which the first $f + 1$ actions of $\beta$ are fail, $i \in J$, and no other fail actions occur in $\beta$. Note that, for all $i \in J$, $\beta$ contains no output actions of $P_i$. Assume that in $\beta$, no perform$_{i,c}$ or b$_i$ (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 enables a dummy action in every i* task of every service and register ($s$ is perform or output).

Since $\alpha_0$ 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 $\gamma'$ is obtained from
γ by removing the fail_i action, all dummy actions, and any remaining internal actions of P_i, i ∈ J. Thus, o_0 · γ is a failure-free extension of o_0 that includes decide(v). Since o_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 o_0. Suppose for contradiction that the decide(0)_l action occurs in the prefix o_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 ≠ j, the same decision value 0 appears in the state s_1. But this contradicts the assumption that o_1, which ends in s_1, is 1-valent. So, it must be the case that the decide(0)_l occurs in the suffix γ’.

Now we show how to append essentially the same γ’ after o_1. We know that, for every i ∈ J, γ’ contains no locally controlled action of P_i, and contains no perform_{i,c} or b_{i,c} action (b ∈ resps), for any c ∈ K ∪ R.

By definition of j-similarity, we have:

(a) For every i ∈ J, the state of P_i is the same in s_0 and s_1.
(b) For every c ∈ K ∪ R,
   1. The value of v_{0,c} is the same in s_0 and s_1 (that is, in the final states of o_0 and o_1).
   2. For every i ∈ 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 ∈ K ∪ R,
   1. The value of v_{0,c} is the same in s_0 and s_1.
   2. For every i ∈ J, if γ’ contains any perform_{i,c} or b_{i,c} (b ∈ 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 o_1, resulting in a failure-free extension of o_1 that includes decide(0)_l.3 But o_1 is 1-valent — a contradiction. □

Similarly, we define the notion of k-similar states:
Let k ∈ K, and let s_0 and s_1 be states of C. Then s_0 and s_1 are k-similar if the following conditions hold:

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

(2) For every c ∈ (K - {k}) ∪ R, the state of C is the same in s_0 and s_1.

Lemma 7 Let k ∈ K. Let o_0 and o_1 be finite failure-free input-first executions, s_0 and s_1 the respective final states of o_0 and o_1. Suppose that s_0 and s_1 are k-similar. If o_0 and o_1 are univalent, then they have the same value.

Proof: Fix k, o_0, α_1, s_0, and s_1 as in the hypotheses of the lemma. By contradiction, suppose (without loss of generality) that o_0 is 0-valent and o_1 is 1-valent. Let J ⊆ I be any set of indices such that |J| = f + 1, and, if |J| ≤ f + 1, then J ∩ J_k ⊆ J, whereas if |J| > f + 1, then J ∩ J_k.

Consider a fair extension of o_0, α_0 β, in which the first f + 1 actions of β are fail_i, i ∈ J, and no other fail actions occur in β. Note that, for all i ∈ J, β contains no output actions of i. Assume that in β, no perform_{i,k} or b_{i,k} action (b ∈ 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 o_0 is a failure-free input-first execution, the resulting extension α_0 β 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 α_0 β, which we denote by α_0 γ, that includes decide(v)_l for some l ∈ I - J and v ∈ {0, 1}. We know that decide(0)_l occurs in the suffix γ, rather than in the prefix o_0, by an argument similar to that in the proof of Lemma 6.

Now construct α_0 γ, where γ is obtained from γ by removing all the fail_i actions, i ∈ J, and all dummy actions. Thus, α_0 γ is a failure-free extension of α_0 that includes decide(v)_l. Since o_0 is 0-valent, v must be equal to 0.

Now we show how to append essentially the same γ’ after o_1. By definition of k-similarity, we have:

(a) For every i ∈ I, the state of P_i is the same in s_0 and s_1.
(b) For every c ∈ (K - {k}) ∪ R, the state of C is the same in s_0 and s_1.

Thus:

(c) For every c ∈ K ∪ R, if γ’ contains any perform_{i,c} or b_{i,c} actions of S_k, then the state of C is the same in s_0 and s_1.

By properties (a) and (c), it follows that it is possible to append “essentially” the same γ’ after o_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 \( G(C) \) 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 \( \alpha, \alpha', \alpha_0, \) and \( \alpha_1, \) and let \( e \) and \( e' \) be the two tasks involved in the hook, as shown. Since \( \alpha_0 \) and \( \alpha_1 \) 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, \( \alpha_0 \) is 0-valent, whereas \( \alpha' \) has a 1-valent failure-free extension \( \alpha_1 \) — a contradiction.

Claim 1 and Lemma 2 imply that \( e' \) is enabled from \( e(s). \)

Claim 2: \( \text{participants}(e,s) \cap \text{participants}(e',s) \neq \emptyset. \)

Suppose for contradiction that \( \text{participants}(e,s) \cap \text{participants}(e',s) = \emptyset. \) Therefore, the two tasks commute, that is, \( e'(e(s)) = e'(e'(s)). \) In other words, \( e'(s_0) = s_1 \) — a contradiction.

Since \( \text{participants}(e,s) \cap \text{participants}(e',s) \neq \emptyset, \) 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 \( P_i \in \text{participants}(e,s) \cap \text{participants}(e',s). \)

Suppose for contradiction that \( P_i \in \text{participants}(e,s) \cap \text{participants}(e',s). \) Then the two actions \( \text{action}(e,s) \) and \( \text{action}(e',s) \) involve only \( P_i \) and the buffers \( \text{buffer}(c), c \in K \cup R. \) Furthermore (since the same task \( e \) is used), the action \( \text{action}(e',s) \) also involves only \( P_i \) and the buffers \( \text{buffer}(c), c \in K \cup R. \) But then the states \( s_0 \) and \( s_1 \) can differ only in the state of \( P_i \) and in the values of \( \text{buffer}(c), c \in K \cup R. \) This implies that \( s_0 \) and \( s_1 \) are \( i \)-similar — a contradiction.

Claim 4: There does not exist \( k \in K \) such that \( S_k \in \text{participants}(e,s) \cap \text{participants}(e',s). \)

Suppose for contradiction that \( S_k \in \text{participants}(e,s) \cap \text{participants}(e',s). \) There are four possibilities:

1. \( \text{participants}(e,s) = \text{participants}(e',s) = \{S_k\}. \)
   Then \( e \) and \( e' \) must be \( \text{perform} \) 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 \( S_k. \) Then the two tasks commute, that is, \( e'(s_0) = s_1 \) — a contradiction.

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

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

4. For some \( i, j \in I, \) \( \text{participants}(e,s) = \{S_k, P_i\} \) and \( \text{participants}(e',s) = \{S_k, P_j\}. \)
   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 \( \text{action}(e,s) \) and \( \text{action}(e',s) \) access the same buffer, one action inserts an item and the other removes an item. Hence the actions commute.

Claim 5: There does not exist \( r \in R \) such that \( S_r \in \text{participants}(e,s) \cap \text{participants}(e',s). \)

Suppose for contradiction that \( S_r \in \text{participants}(e,s) \cap \text{participants}(e',s). \) There are four possibilities:

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

   (a) \( \text{action}(e,s) \) and \( \text{action}(e',s) \) both perform read operations.

   Then the two tasks commute, so \( e'(s_0) = s_1 \) — a contradiction.

   (b) \( \text{action}(e,s) \) performs a write operation.

   Then states \( s_0 \) and \( s_1 \) can differ only in the value of \( \text{inv-buffer}(j)_r \) and \( \text{resp-buffer}(j)_r: \) in \( s_1, \) an invocation is missing from \( \text{inv-buffer}(j)_r \) and an extra response appears at the end of \( \text{resp-buffer}(j)_r, \) with respect to \( \text{inv-buffer}(j)_r \) and \( \text{resp-buffer}(j)_r, \) 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(s).

Then c'(s_0) and s_1 differ only in the value of resp_buffer(i), (different read responses may be appended at the end). So c'(s_0) and s_1 are k-similar — a contradiction.

2. For some i ∈ I, \(\text{participants}(e,s) = \{S_i, P_1\}\) and \(\text{participants}(e',s) = \{S_i\}\).

Then the two tasks commute, so c'(s_0) = s_1 — a contradiction.

3. For some i ∈ I, \(\text{participants}(e',s) = \{S_i, P_1\}\) and \(\text{participants}(e,s) = \{S_i\}\).

Again, the two tasks commute, so c'(s_0) = s_1 — a contradiction.

4. For some i, j ∈ I, \(\text{participants}(e,s) = \{S_i, P_1\}\) and \(\text{participants}(e',s) = \{S_j\}\).

By Claim 3, we know that i ≠ j. Then the two tasks commute, so c'(s_0) = s_1 — a contradiction.

Now Claims 3, 4, and 5 together imply that \(\text{participants}(e,s) \cap \text{participants}(e',s) = \emptyset\). 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 · (f + 1) principal processes can fail: f' < s · (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 · 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 · (f + 1). Thus, we have s = ⌊(f'+1)/(f+1)⌋ 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 ≤ k < m, k ≤ f ≤ m−1, 1 ≤ f' ≤ 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

\[
k' \geq k \cdot \left\lceil \frac{f'}{f+1} \right\rceil + \min(k, (f'+1)\mod(f+1)).
\]

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 T, the service has a service type parameter U, which is a tuple <V, V₀, invs, resps, glob, δ₁, δ₂, δ₃>, where V and V₀ 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 δ₁, δ₂, δ₃ are three transition relations.

Here, δ₁ is a total binary relation from invs × J × V to (the set of mappings from J to finite sequences of resps) × V. It is used to map an invocation at the head of a particular inv−buffer, and the current value for invl, to a set of results, each of which consists of a new value for invl and sequences of responses to be added to any or all of the resp−buffers. δ₂ is a total binary relation from J × V to (the set of mappings from J to finite sequences of resps) × V. It is used to map a particular endpoint and value of invl to a set of results, defined as above. Finally, δ₃ is a total binary relation from V to (the set of mappings from J to finite sequences of resps) × V. It is used to map a value of invl 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 U, endpoint set J, and index k, provided that it implements the canonical f-resilient failure-oblivious service S of type 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 U = <V, V₀, invs, resps, glob, δ₁, δ₂, δ₃>. Here, V consists of a single msgs queue, containing messages that have been totally ordered, together with their sources (Figure 5). V₀ indicates that this queue is initially empty.

The invocation set invs is {cast(m) : m ∈ M}. The response set resps is {recv(m, i) : m ∈ M, i ∈ J}, (recv(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, glob = {g}, δ₁, the relation describing the transitions that process invocations from inv−buffers, is defined in Figure 6.

This code processes the first element of inv−buffer(i) by adding it to the end of the sequence stored in msgs. (Formally, δ₁((a, i, v), (B, v')) holds if and only if δ₁((a, i, v), (B, v')) holds if a = cast(m), v'.msgs is the result of adding (m, i) to the end of v.jms, and B(j) is empty for all j.)

δ₂ is the identity relation, indicating that no other processing is done on behalf of i. Relation δ₃ is defined in Figure 7:

(Formally, δ₃(v, (B, v')) holds if either (a) v.jms is nonempty, (m, i) = head(v.jms), v'.msgs = tail(v.jms), and for every j ∈ J, B(j) is the sequence consisting of the single element recv(m, i), or (b) v.jms 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 Sk.

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

Lemma 2: We have added the c-compute and g-compute tasks to the definition of a service, Figure 4. These are defined using total transition relations δ₂ and δ₃. Since these are total relations, we see from Figure 4 that these tasks are always enabled. Hence Lemma 2 still holds.
Canonical Failure Oblivious Service $(l, J, f, k)$, where $l = (V, V_0, \text{msg}, \text{resps}, \text{globals}, \delta_1, \delta_2, \delta_3, \delta_4)$

**Signature:**

**Inputs:**
- $a_{i,k}, a \in \text{msg}, i \in J$
- $\text{fail}, i \in J$

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

**Internals:**
- $\text{perform}_{i,k}, i \in J$
- $\text{compute}_{i,k}, i \in J$
- $\text{dummy} \cdot \text{compute}_{i,k}, * \in \{ \text{perform}, \text{compute}, \text{output} \}, i \in J$
- $\text{compute}_{g,k}, g \in \text{globals}$
- $\text{dummy} \cdot \text{compute}_{g,k}, g \in \text{globals}$

**State Components:**
As for canonical atomic object.

**Transitions:**

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

**Internal:** $\text{perform}_{i,k}$
Precondition:
- $a = \text{head}(\text{inv} \cdot \text{buffer}(i))$
- $\delta_1 ((a, v, B(a, v)))$

**Effect:**
- Remove head of $\text{inv} \cdot \text{buffer}(i)$
- $\text{val} \leftarrow v$
- For $j \in J$ do
  - add $B(j)$ to end of $\text{resp} \cdot \text{buffer}(j)$

**Internal:** $\text{compute}_{i,k}, i \in J$
Precondition:
- $\delta_2 ((v, \text{val}, B(a, v)))$

**Effect:**
- $\text{val} \leftarrow v$
- For $j \in J$ do
  - add $B(j)$ to end of $\text{resp} \cdot \text{buffer}(j)$

**Internal:** $\text{compute}_{g,k}, g \in \text{globals}$
Precondition:
- $\delta_3 (\text{val}(B(a, v)))$

**Effect:**
- $\text{val} \leftarrow v$
- For $j \in J$ do
  - add $B(j)$ to end of $\text{resp} \cdot \text{buffer}(j)$

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

**Input:** $\text{fail}$
As for canonical atomic object.

**Internal:** $\text{dummy} \cdot \text{compute}_{i,k}, i \in J$
As for canonical atomic object.

**Internal:** $\text{dummy} \cdot \text{compute}_{g,k}, g \in \text{globals}$
Precondition:
- $|\text{failed}| > f$

**Effect:**
- None

**Tasks:**
- For every $i \in J$:
  - $\text{perform} \{ \text{perform}_{i,k}, \text{dummy} \cdot \text{perform}_{i,k} \}$
  - $\text{compute} \{ \text{compute}_{i,k}, \text{dummy} \cdot \text{compute}_{i,k} \}$
  - $\text{output} \{ b_{i,k} : b \in \text{resps} \} \cup \{ \text{dummy} \cdot \text{output}_{i,k} \}$
- For every $g \in \text{globals}$:
  - $\text{compute} \{ \text{compute}_{g,k}, \text{dummy} \cdot \text{compute}_{g,k} \}$

**Figure 4:** A canonical failure-oblivious service.

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

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

**Internal:** $\text{perform}_{i,k}$
Precondition:
- $\text{send}(m) = \text{head}(\text{inv} \cdot \text{buffer}(i))$

**Effect:**
- Remove head of $\text{inv} \cdot \text{buffer}(i)$
- Add $(m, i)$ to msg$:$

**Figure 6:** Relation $\delta_1$ 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 $\text{perform}_{i,k}$ actions with a reference to $\text{perform}_{i,k}$ or $\text{compute}_{i,k}$ or $\text{compute}_{g,k}$ actions. We provide a complete proof in Appendix B.

Lemma 7: Since service $S_k$ is “silent” along $\gamma$, the change in its definition does not affect the proof. The other services have the same behavior along $\gamma$ and $\gamma'$, 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$(e, s) = \text{participants}(e', s) = \{ S_k \})$ must be modified by replacing every reference to $i \rightarrow \text{perform}$ or $i \rightarrow \text{compute}$ or $i \rightarrow \text{compute}$ tasks with a reference to $i \rightarrow \text{perform}$ or $i \rightarrow \text{compute}$ or $i \rightarrow \text{compute}$ 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.
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 inus sets, that is, their only inputs are fail_i actions.

7.2.1 Perfect Failure Detector 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' ⊆ 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: δ_2(i, failed) simply puts a suspect response containing the current failed set into i’s response buffer (Figure 9).

7.2.2 Eventually Perfect Failure Detector ◇P

Again, responses are of the form suspect(J'), J' ⊆ 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 ◇P is presented in Figure 10.

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

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

Figure 9: Relation δ_2 in P.

Figure 10: The composition of val in ◇P.
The global task set $\text{glob} = \{g_1, g_2\}$. Task $g_1$ is responsible for setting $\text{mode}$ to $\text{imperfect}$ while task $g_2$ sets it to $\text{perfect}$. The interesting transition definitions are presented in Figure 11.

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

**Internal**: $\text{compute}_{g_1,k}$
Precondition:  
true
Effect:  
if $\text{failed} \neq \text{oldfailed}$ then  
$\text{mode} := \text{imperfect}$  
$\text{oldfailed} := \text{failed}$

**Internal**: $\text{compute}_{g_2,k}$
Precondition:  
true
Effect:  
if $\text{failed} = \text{oldfailed}$ then  
$\text{mode} := \text{perfect}$

Figure 11: Internal transitions in $\Diamond P$.

### 7.2.3 Eventual Leader Service $\Omega$

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

**Components of $\text{val}$**

- $\text{mode} \in \{\text{perfect}, \text{imperfect}\}$, initially $\text{imperfect}$
- $\text{oldfailed} \subseteq J$, initially $\emptyset$
- $\text{leader} \in J \cup \{\bot\}$, initially $\bot$

Figure 12: The composition of $\text{val}$ in $\Omega$.

We again use two global tasks $g_1, g_2$. Now $g_1$ sets $\text{mode}$ to $\text{imperfect}$ and removes any choice of leader, while $g_2$ sets $\text{mode}$ to $\text{perfect}$ 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 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 $\alpha_0$ and $\alpha_1$ 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 $a_0$ and $a_1$ 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 $a_0$ and $a_1$.

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 $a_0$, $a_0 \cdot \gamma'$, such that (1) $\gamma'$ includes $\text{decide}(f)_l$, for some $l \in I-J$; (2) $\gamma'$ includes no locally controlled step of process $P_j$, nor any $\text{perform}_j$, $\text{compute}_j$, or $\text{output}_j$ step for any service or register (respectively), $\gamma'$ includes no locally controlled step of service $S_j$; (3) $\gamma'$ includes no locally controlled step of any failure-aware service. Thus, $\gamma'$ is essentially applicable to $a_1$ — a contradiction with the assumption that $a_0$ and $a_1$ 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 services are “silent” along $\gamma$, 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 $\text{perform}_{i,k}$, $\text{compute}_{i,k}$, and $\text{compute}_{q,k}$, in particular, their ability to observe the set of failed processes, makes no difference. Hence $\gamma'$ can be appended after $a_1$ 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 “silent” 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 fail of failed processes is empty. Thus the different definition for actions $\text{perform}_{i,k}$, $\text{compute}_{i,k}$, and $\text{compute}_{q,k}$, 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, $0 \leq f < n-1$. 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


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.

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 \( T \) be a sequential type. Then there exists an \( f \)-resilient \( n \)-process implementation of an atomic object of type \( T \) from \( f \)-resilient \( n \)-process consensus objects and reliable registers.

The following result is shown in [16]:

Lemma 15 Let \( n \) be an 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-\( L \) 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-\( L \) 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 $\alpha$ of $A$ in which process $i$ is faulty and is declared to be faulty. Now we consider an execution $\alpha'$ that is identical to $\alpha$ except that $\alpha'$ includes no $fail_i$ event ($i$ is just slow to take steps in $\alpha'$). Clearly, $\alpha'$ is also a finite execution of $A$, since registers and consensus objects are failure-oblivious. Thus, in $\alpha'$, a process is declared faulty without having failed—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 $a_0$ and $a_1$ be finite failure-free input-first executions, $s_0$ and $s_1$ the respective final states of $a_0$ and $a_1$. Suppose that $s_0$ and $s_1$ are $j$-similar. If $a_0$ and $a_1$ 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 $j$, $a_0$, $a_1$, $s_0$, and $s_1$ as in the hypotheses of the lemma, and suppose (without loss of generality) that $a_0$ is $0$-valent and $a_1$ 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 $a_0$, $a_1 \cdot \beta$, in which the first $f + 1$ actions of $\beta$ are fail$_i$, $i \in J$, and no other fail actions occur in $\beta$. Note that, for all $i \in J$, $\beta$ contains no output actions of $P_i$. Assume that in $\beta$, no perform$_i$, compute$e_i$, or bis$e_i$ action of any $2^* = 1$-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 $\beta$, no compute$_g$, 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$\cdot$compute$_g$ action in every $g$-compute task of every failure-oblivious service $c$.

Since $a_0$ is a failure-free input-first execution, the resulting extension $a_0 \cdot \beta$ 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 \(a_0 \cdot \beta\), which we denote by \(a_0 \cdot \gamma\), that includes \(\text{decide}(v)_l\) for some \(l \notin J\) and \(v \in \{0, 1\}\). Construct \(a_0 \cdot \gamma\), where \(\gamma\) is obtained from \(\gamma\) by removing the \(\text{fail}_i\) action, all dummy actions, and any remaining internal actions of \(P_i\), \(i \in J\). Thus, \(a_0 \cdot \gamma\) is a failure-free extension of \(a_0\) that includes \(\text{decide}(v)_l\). Since \(a_0\) is \(0\)-valent, \(v\) must be equal to 0.

We claim that \(\text{decide}(0)_l\) occurs in the suffix \(\gamma\), rather than in the prefix \(a_0\). Suppose for contradiction that the \(\text{decide}(0)_l\) action occurs in the prefix \(a_0\). 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 \(a_1\), which ends in \(s_1\), is \(1\)-valent. So, it must be that the \(\text{decide}(0)_l\) occurs in the suffix \(\gamma\).

Now we show how to append essentially the same \(\gamma\) after \(a_1\). We know that, for every \(i \in J\), \(\gamma\) contains no locally controlled action of \(P_i\), and contains no \(\text{perform}_{i,c}, \text{compute}_{i,c}\), or \(b_{i,c}\) actions, 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 \(\text{vol}_c\) is the same in \(s_0\) and \(s_1\) (that is, in the final states of \(a_0\) and \(a_1\).
2. For every \(i \in J_\neq J\), the value of \(\text{buffer}(i)_c\) is the same in \(s_0\) and \(s_1\).

Thus:

(c) If \(\gamma\) 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 \(\text{vol}_c\) is the same in \(s_0\) and \(s_1\).
2. For every \(i \in J_\neq J\), if \(\gamma\) contains any \(\text{perform}_{i,c}, \text{compute}_{i,c}\), or \(\text{output}_{i,c}\) actions, then \(i \notin J\), and so the value of \(\text{buffer}(i)_c\) is the same in \(s_0\) and \(s_1\).

Finally, we note that the presence of \(\text{compute}_{g,c}\) does not invalidate the argument. A \(\text{compute}_{g,c}\) cannot refer to or modify any input buffers. The precondition of \(\text{compute}_{g,c}\) depends only on \(\text{vol}_c\), and so the same \(\text{compute}_{g,c}\) actions can be applied in \(\gamma\) after \(a_0\), and they can add the same items to the output buffers. Thus for \(i \notin J\) the sequence of values of \(\text{buffer}(i)_c\) takes along \(\gamma\) after \(a_0\) and \(\gamma\) after \(a_1\) are the same.

It follows that it is possible to append "essentially" the same \(\gamma\) after \(a_1\), resulting in a failure-free extension of \(a_1\) that includes \(\text{decide}(0)_l\).\(^6\)

But \(a_1\) is \(1\)-valent — a contradiction. \(\square\)

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

**Lemma 7** Let \(k \in K\). Let \(a_0\) and \(a_1\) be finite failure-free input-first executions, \(s_0\) and \(s_1\) the respective final states of \(a_0\) and \(a_1\). Suppose that \(s_0\) and \(s_1\) are \(k\)-similar. If \(a_0\) and \(a_1\) are univalent, then they have the same valence.

**Proof:** Fix \(k, a_0, a_1, s_0,\) and \(s_1\) as in the hypotheses of the lemma. By contradiction, suppose (without loss of generality) that \(a_0\) is \(0\)-valent and \(a_1\) 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 \(a_0\), \(a_0 \cdot \beta\), in which the first \(f + 1\) actions of \(\beta\), \(i \in J\), and no other \(\text{fail}\) actions occur in \(\beta\). Note that, for all \(i \in J\), \(\beta\) contains no output actions of \(i\). Assume that in \(\beta\), no \(\text{perform}_{i,k}\) or \(b_{i,k}\) or \(\text{compute}_{i,k}\) or \(\text{output}_{i,k}\) action \((b \in \text{resp}\), \(g \in \text{glob}\) of \(S_k\) occurs; we may assume this because the \(f + 1\) \(\text{fail}\) actions enable dummy actions in all tasks of \(S_k\).

Since \(a_0\) is a failure-free input-first execution, the resulting extension \(a_0 \cdot \beta\) 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 \(a_0 \cdot \beta\), which we denote by \(a_0 \cdot \gamma\), that includes \(\text{decide}(v)_l\) for some \(l \notin J\) and \(v \in \{0, 1\}\). We know that \(\text{decide}(0)_l\) occurs in the suffix \(\gamma\), rather than in the prefix \(a_0\), by an argument similar to that in the proof of Lemma 6.

Now construct \(a_0 \cdot \gamma\), where \(\gamma\) is obtained from \(\gamma\) by removing all the \(\text{fail}\) actions, \(i \in J\), and all dummy actions. Thus, \(a_0 \cdot \gamma\) is a failure-free extension of \(a_0\) that includes \(\text{decide}(v)_l\). Since \(a_0\) is \(0\)-valent, \(v\) must be equal to 0.

Now we show how to append essentially the same \(\gamma\) after \(a_1\). By definition of \(k\)-similarity, we have:

(a) For every \(i \in J\), the state of \(P_i\) is the same in \(s_0\) and \(s_1\).
(b) For every \(c \in (K \setminus \{k\}) \cup R\), the state of \(S_c\) is the same in \(s_0\) and \(s_1\).

Thus:

\(^6\)Really, we are appending another execution fragment \(\gamma'\) after \(a_1\) — one that looks the same to all the processes and service tasks that take steps in \(\gamma\).
(c) For every \( c \in K \cup R \), if \( \gamma' \) contains any \( \text{perform}_{i,c} \) or \( \text{compute}_{j,k} \) or \( \text{compute}_{g,k} \) 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 \( \gamma' \) after \( \alpha_1 \), (differing only in the state of \( S_k \)) resulting in a failure-free extension of \( \alpha_1 \) that includes \( \text{decide}(0) \). But \( \alpha_1 \) is 1-valent — a contradiction. \( \square \)

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 \( P_j \) 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 remove 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 \( S_k \in \text{participants}(e, s) \cap \text{participants}(e', s) \).

Suppose for contradiction that \( S_k \in \text{participants}(e, s) \cap \text{participants}(e', s) \). There are four possibilities:

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

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

   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 \( i \in I \), \( \text{participants}(e', s) = \{ S_k, P_i \} \) and \( \text{participants}(e, s) = \{ S_k \} \).

   Hence \( \text{action}(e, s) \) is one of \( \text{perform}_{i,k} \), \( \text{compute}_{i,k} \), or \( \text{compute}_{g,k} \) where \( j \in J_k, g \in \text{glob} \).

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

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

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