Boosting Fault-Tolerance in Asynchronous Message Passing Systems is Impossible

Paul Attie[1], Nancy Lynch[2], and Sergio Rajsbaum[3]

1. College of Computer Science, Northeastern University, attie@ccs.neu.edu
2. MIT Laboratory for Computer Science, lynch@theory.lcs.mit.edu
3. Mathematics Institute, Universidad Nacional Autonoma de Mexico (UNAM), rajsbaum@matem.unam.mx

December 21, 2002

Abstract

We show that it is impossible to "boost" the level of fault-tolerance of a system solving consensus by combining less fault-tolerant components into a more fault-tolerant system. To do this, we consider an asynchronous distributed computing model in which a known set of processes interact in two ways: by using reliable point-to-point channels, and by accessing shared services. Each of the shared services is connected to a subset of all the processes.

Our boosting impossibility result is: for any $f \geq 1$, the consensus problem is unsolvable in this model in the presence of up to f process stopping failures, if each of the shared services is assumed to tolerate only $f-1$ process failures. This result holds regardless of the types of the shared services and the pattern of connectivity of processes and services. In particular, it is impossible to construct a protocol to solve the consensus problem for f process failures using any number of consensus services that tolerate $f-1$ process failures.

Interestingly, it is possible to boost the level of a system solving problems easier than consensus. For example, we show that the k-consensus problem is solvable for $2k-1$ failures using only (consensus) services that tolerate only 1 failure apiece.

1 Introduction

It is generally accepted that large distributed systems should be constructed from building blocks (such as middleware-provided services) that interact with each other through well-defined interfaces. Large systems must also tolerate a variety of types of failures. Establishing fault-tolerance properties of a large system is difficult, as many scenarios have to be considered. A particularly desirable approach is to "boost" the level of fault-tolerance by combining less fault-tolerant components into a more fault-tolerant system. It is plausible that this might be achieved using techniques such as quorums, replication, and redundancy.

In this paper, we demonstrate a fundamental limitation on this approach. Namely, we investigate the possibility of fault-tolerance boosting for implementing a consensus service tolerant to f stopping failures from underlying "subservices" that are tolerant to $f-1$ stopping failures. We show that, in the setting of purely asynchronous message passing, such fault-tolerance boosting cannot be achieved, for any type of underlying services. That is, the availability of any set of distributed services, each of which tolerates up to $f-1$ stopping failures, is insufficient to construct a consensus protocol that tolerates f failures.

In more detail, we consider a set of asynchronous processes of which f can fail by stopping, communicating with each other by sending messages through reliable point to point channels. In addition, there is a set of services through which they can communicate implicitly. A process can invoke operations of a service by sending a message to one of its ports, and eventually get a response from the service. A process can invoke multiple operations on a service, and concurrently on other services. But before issuing a new operation on the same service, it must first wait for a response to the current invocation. Each service has a fixed set of "ports" and each port is hardwired to one process, where it receives invocations and returns responses to the corresponding process. Each service has some degree of fault tolerance, say f, which represents the number of (hardwired) processes accessing it that could cause it to crash. This is intended to reflect the idea that services are implemented by distributed algorithms, which run at a number of locations, represented by ports. The failure really affects the location, causing not only the failure of the process hardwired to the corresponding port, but also the failure of that part of the distributed implementation of the service which resides at that location. If a sufficient number ($> f$) of locations of a distributed implementation fail, then the implementation itself will fail. Note that this idea does not in any way prevent the use of arbitrary oracles in the implementation of a service, e.g., such as failure detectors or powerful hardware concurrent objects.

Notice that, except for the failure behavior, our services are just like the linearizable typed shared objects usually considered in the literature e.g. [Her91, CJT94, Jay97, LH00]. The services usually considered in the literature do not fail at all. There are only two papers we are aware of that consider services that can fail, [JCT98] and [AGMT95], but these papers assume the services are not implemented by the processes. In contrast to our model, the failures of the services and of the processes are not correlated in those two papers. We discuss this further in the Related Work section below.

Our impossibility result says that it is impossible to build a consensus service tolerating f failures from services that tolerate less than f failures, independently of the number of such services, how powerful they are, or in what way they are accessed by the processes. Thus, for example, a strategy in which multiple instances of $(f-1)$-fault-tolerant services are used by different subsets of the processes in the system, cannot work. Methods based on splitting up processes, or divide and conquer, also cannot work. In particular, our result holds when the underlying services include

consensus services tolerant to $f-1$ stopping failures.

It is important to study consensus implementability because it is such a fundamental problem in distributed computing. In particular, there is Herlihy's [Her91] universality result for services that do not fail: it is possible to design a wait-free implementation of a service of any type, shared by n processes, using only consensus services with n ports and registers. Our boosting impossibility result shows a limitation on this universality result when services can fail.

Our impossibility holds for consensus implementability, but not for implementability of weaker problems. Our second result is that it is *is* possible to boost the level of a system solving problems easier than consensus, like k-consensus. In this problem processes have to agree on at most k different values; thus, k-consensus reduces to consensus when $k = 1$. We present a simple algorithm (generalizing the one in [HR94, HR00]) that solves k-consensus and tolerates f failures using k'-consensus services that tolerate f' less than f failures, for various values of k' and f'. For example, k-consensus is solvable for $2k-1$ failures using only (consensus) services that tolerate only 1 failure apiece.

Related work. Our main result is the impossibility of solving consensus f-resiliently using $f-1$-resilient services in an asynchronous system. There is a lot of work that studied the feasibility of implementing f-tolerant consensus as a function of the available components in the asynchronous system. The "components" can be simple message transmission channels or shared read/write registers, but also more powerful objects, perhaps implemented in hardware such as test&set or implemented with timeouts such as failure detectors, or even combinations of different kinds of objects. A *typed shared object* used in many papers is what we call a service, i.e., it has (i) a number of ports; (ii) a set of states of the object (or values as we call them); (iii) the set of operations that processes may apply through its ports; (iv) the behavior of the object in terms of a transition relation δ, and is assumed to be linearizable. Except that the usual assumption is that the components themselves are reliable.

Work that assumed that the available components are the most basic ones is [FLP85] for just message transmission, and [LAA87, Her91] for shared read/write registers, and proved that it is impossible to solve f-tolerant consensus using only these simple components. That is, the available components, either channels or registers never fail. Since a consensus protocol that tolerates zero crash faults is trivial, our result generalizes that of [FLP85], which is a special case, for $f = 1$. Indeed, our proof technique is a generalization of the one in [FLP85]. The main difference is the idea of modelling the services. This introduces many more scenarios to deal with in the proof. Also, our events are much finer grain: in FLP, in one event a process receives a message, makes a local state change, and also sends any finite number of messages. Our events are I/O automata actions in the model of distributed systems with services. So, for example, a process receiving a message can only make a local state change, it cannot perform any output of any kind in the same event.

Other papers consider more general and powerful base objects (again that never fail), and investigate when they can be used to solve consensus. For example, [LH00] ask the question for $f = 1$: Let $n \geq 3$ and $\mathcal{S}$ be a set of object types that can be used to solve one-resilient consensus among n processes. Can $\mathcal{S}$ always be used to solve one-resilient consensus among $n-1$ processes? Many papers consider the other extreme, of $f = n-1$ and deal with the robustness question posed in [Jay97]: can you combine objects of type T and T' that cannot be used to solve wait-free consensus each one by themselves in such a way as together solve wait-free consensus?

Other papers relate implementations for different number of processes based on the same fault-tolerance level f. Specifically, [CJT94] show for all $n > f \geq 2$ and all sets $\mathcal{S}$ of shared object types

(that include simple read/write registers) there is a f-resilient solution to n-process consensus using objects of types in $\mathcal{S}$ if and only if there is a f-resilient solution to $(f+1)$-process consensus using objects of types in $\mathcal{S}$. And [BGLR01] for k-set consensus: if there is a f-resilient implementation of n-ported f-set consensus from registers then there is a f-resilient implementation of $f+1$-ported f-set consensus from registers.

Thus, our question is orthogonal to the concerns of these previous works: while they assume reliable components, we consider components that are less reliable, i.e. we ask what problems can be solved in an f-resilient manner using components that tolerate less than f failures. We know of two papers that do consider shared objects that may fail. Afek, Greenberg, Merritt, Taubenfeld [AGMT95] study wait-free implementations using objects that can fail by returning the wrong value for a response. And more closely related to our work is [JCT98] that consider base objects that may fail by not responding (both [JCT98] and [AGMT95] consider other types of failures, like wrong values returned, less related to our work). In their model any number of processes may fail, and at most t base objects may fail. When an object fails, it stops responding. They have an impossibility result for solving consensus for two processes tolerating even one nonresponsive-faulty service, and even if that service can be nonresponsive wrt only one predetermined process. This proof works by a reduction from [LAA87]. This result is orthogonal to ours: the failures of the services in their model are unrelated to the failures of the processes, while in our model, services can fail only due to failures of processes. Thus, if no process fails, in our model we know no service will fail, while in such a situation in their model still services could fail. On the other hand, they know that at most one service will fail, while in ours there is no bound: if one service will fail due to too many processes failing, all the services with the same processes associated can also fail.

Our main concern in this paper is on the implementation of consensus. Recall that Herlihy [Her91] has shown that any object can be implemented using consensus. Thus consensus is at the top of a hierarchy. As mentioned above, our impossibility result does not hold for objects weaker than consensus.

The paper is organized as follows. Section 2 gives technical preliminaries. Section 3 gives our model of a distributed system, and defines the consensus problem. Section 4 presents our impossibility result for consensus. Section 5 describes the contrasting result for k-set consensus. Section 6 discusses directions for further research and concludes. Appendix A presents some technical background.

2 Modeling Preliminaries

2.1 Basic underlying model of concurrent computation

We use the I/O automaton model [Lyn96, chapter 8] as our underlying model for concurrent computation. We assume the terminology of [Lyn96, chapter 8]. An I/O automaton A is *deterministic* iff, for each task t of A, and each state s of A, there is at most one transition (s, a, s') such that $a \in t$.

2.2 Variable types

We define the notion of a "variable type", in order to describe allowable sequential behavior of services. The definition used here is a generalization of the one in [Lyn96, chapter 9]; the gener-

alization allows nondeterminism in the choice of the initial state and the next state. Namely, a *variable type* $\mathcal{T} = \langle V, V_0, invs, resps, \delta \rangle$ consists of:

- V, a nonempty set of states of the variable, called *values*,
- $V_0 \subseteq V$, a nonempty set of *initial values*,
- $invs$, a set of *invocations*,
- $resps$, a set of *responses*, and
- δ, a subset of $(invs \times V) \times (resps \times V)$ that is "total", in the sense that, for every $(a, v) \in invs \times V$, there is at least one $(b, v') \in resps \times V$ such that $((a, v), (b, v')) \in \delta$.

A *deterministic* variable type is one in which δ is a mapping, i.e., for every $(a, v) \in invs \times V$, there is *exactly* one $(b, v') \in resps \times V$ such that $((a, v), (b, v')) \in \delta$.

The reason for generalizing the notion of a variable type to allow nondeterminism is that we want to make our notion of "service", defined below, as general as possible. In particular, we want to include the problem of k-consensus, which can be specified using a nondeterministic variable type, in our consideration.

Example. *Read/write variable type:* Here, V is some arbitrary set of "values," $V_0 = V$, $invs = \{read\} \cup \{write(v) : v \in V\}$, $resps = V \cup \{ack\}$, and δ is defined to include the following pairs: $((read, v), (v, v))$ for $v \in V$, and $((write(v), v'), (ack, v))$ for $v, v' \in V$. □

Example. *Consensus variable type:* Here, V is the set of subsets of $\{0, 1\}$ having at most one element, $V_0 = \emptyset$, $invs = \{init(v) : v \in \{0, 1\}\}$, $resps = \{decide(v) : v \in \{0, 1\}\}$, and δ is defined to include the following pairs:
$((init(v), \emptyset), (decide(v), \{v\}))$ for $v \in V$, and $((init(v), \{v'\}), (decide(v'), \{v'\}))$ for $v, v' \in V$. □

Example. *k-consensus variable type:* Here, V is the set of subsets of $\{0, 1, \ldots, k\}$ having at most k elements, $V_0 = \emptyset$, $invs = \{init(v) : v \in \{0, 1\}\}$, $resps = \{decide(v) : v \in \{0, 1\}\}$, and δ is defined to include the following pairs: $((init(v), W), (decide(v'), W \cup \{v\}))$ for $|W| < k$, $v' \in W \cup \{v\}$, and $((init(v), W), (decide(v'), W)$ for $|W| = k$, $v' \in W$.
Thus, the first k values get remembered, and all operations return one of these first k values. □

2.3 Canonical f-fault-tolerant atomic objects

We now define the notion of canonical f-fault-tolerant atomic object, which describes the allowable concurrent behavior of services. The *canonical f-fault-tolerant atomic object of type $\mathcal{T}$ for endpoint set J and with index k* is given in Figure 1 as an I/O automaton that is parameterized by k, $\mathcal{T}$, J, and f, where these are:

1. A unique index k, drawn from some index set K,
2. An underlying variable type $\mathcal{T} = \langle V, V_0, invs, resps, \delta \rangle$, which defines the sequential behavior of the object,
3. A set of "endpoints" J, and
4. The required degree of fault-tolerance f.

A canonical atomic object accommodates concurrent invocations by different processes, i.e., between an invocation from and response to a particular process, the invocations of other processes may arrive and be processed. The use of a set of endpoints allows different services to be connected to different sets of processes. Thus, J will be a subset of some set I of process indices, which represents all the processes in the system.

Our notion of atomic object generalizes that in [Lyn96, section 13.1.2]. We note the following features of our atomic objects. Each process in J can issue any invocation of the atomic object's underlying variable type, and can (potentially) receive any allowable response. The result of performing an particular operation is nondeterministically selected from all results allowed by the transition relation δ and the current value val of the object. Thus, the object is, in general, inherently nondeterministic in that it can exhibit nondeterminism that is not just due to the nondeterminism of its invocations by different processes.

For every process P_i, $i \in J$, there corresponds a task of the atomic object, which we call an *i-task*. The i-task consists of all the *perform* actions that carry out the operations invoked by P_i, together with all the possible *response* actions giving responses to P_i. In addition, the i-task contains a $dummy_{k,i}$ action, which is enabled when either P_i has failed or more than f processes in J have failed. Thus, by inspecting Figure 1 we see that for every $i \in J$, the task structure requires that the object eventually respond to an outstanding invocation by P_i, unless either P_i has failed or more than f processes in J have failed. In the latter case, the object is allowed to abstain from responding to P_i, since the internal action $dummy_{k,i}$ is enabled, and can be executed to discharge the fairness requirement imposed by the task structure. If more than f processes have failed, then the object is allowed to abstain from responding to any process in J, since $dummy_{k,i}$ is enabled for all $i \in J$. This reflects the idea that the object is f-tolerant; once more than f failures have occurred (amongst processes connected to the object), then the object can itself "fail" by being "silent" forever from that point onwards. That is, we allow the object to violate its liveness property. Note, however, that the object can never violate its safety property, e.g., by returning values inconsistent with the transition relation δ. Note that we also allow the object to be silent if all processes it is connected to (i.e., in J) fail, since $dummy_{k,i}$ is then enabled for all $i \in J$.

2.4 f-fault-tolerant atomic objects

Given a variable type $\mathcal{T}_k$ and set J_k of endpoints, define an I/O automaton U to be a *well-formed environment* for $\mathcal{T}_k$ and J_k if and only if

1. Its outputs are exactly the invocations of $\mathcal{T}_k$ at the endpoints in J_k, and its inputs are exactly the responses of $\mathcal{T}_k$ at the endpoints in J_k, and
2. In every execution of U, for each endpoint $i \in J_k$, there aren't two consecutive invocations at i without an intervening response at i.

An I/O automaton A (a full-blown I/O automaton, with tasks) is said to be an *f-fault-tolerant atomic object* of type $\mathcal{T}_k$, set J_k of endpoints, and index k, if and only if it implements the f-fault-tolerant canonical atomic object S_k of type $\mathcal{T}_k$ for J_k, in the following sense:

1. It has the same input and output actions (including the fail actions).
2. If U is a well-formed environment for $\mathcal{T}_k$ and J_k, then

Canonical Atomic-Object$(k, \langle V, V_0, invs, resps, \delta\rangle, J, f)$

Signature

Input:
$a_{i,k}$, $a \in invs$, the invocations of Atomic-Object$(k, \langle V, V_0, invs, resps, \delta\rangle, J, f)$ by P_i, $i \in J$
$fail_i$, $i \in J$
Output:
$b_{k,i}$, $b \in resps$, the responses of Atomic-Object$(k, \langle V, V_0, invs, resps, \delta\rangle, J, f)$ to P_i, $i \in J$
Internal:
$perform((a,v),(b,v'))_{k,i}$, $a \in invs$, $b \in resps$, $v, v' \in V$, $i \in J$
$dummy_{k,i}$, $i \in J$

State

val, a value in V, initially a value in V_0
$inv-buffer$, a set of pairs (i, a), for a_i an input action
$resp-buffer$, a set of pairs (i, b), for b_i an output action
$failed \subseteq J$, initially empty

Actions

Input $a_{i,k}$
Eff: $inv-buffer \leftarrow inv-buffer \cup \{(i,a)\}$

Internal $perform((a,v),(b,v'))_{k,i}$
Pre: $(i,a) \in inv-buffer \wedge val = v \wedge \delta((a,v),(b,v'))$
Eff: $inv-buffer \leftarrow inv-buffer - \{(i,a)\}$;
$val \leftarrow v'$;
$resp-buffer \leftarrow resp-buffer \cup \{(i,b)\}$

Output $b_{k,i}$
Pre: $\{(i,b)\} \in resp-buffer$
Eff: $resp-buffer \leftarrow resp-buffer - \{(i,b)\}$;

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

Internal $dummy_{k,i}$
Pre: $i \in failed \vee |failed| > f$
Eff: none

Tasks
For every $i \in J$: $\{perform((a,v),(b,v'))_{k,i} : \delta((a,v),(b,v'))\} \cup \{b_i : b \in resps\} \cup \{dummy_{k,i}\}$

Figure 1: I/O automaton for the canonical f-fault-tolerant atomic object with endpoints J and type $\mathcal{T} = \langle V, V_0, invs, resps, \delta\rangle$

(a) Any trace β of $A \times U$ is also a trace of $S_k \times U$. (This should imply that A preserves well-formedness and guarantees atomicity.)

(b) Any fair trace β of $A \times U$ is also a fair trace of $S_k \times U$. (This should imply that the implementation is f-fault-tolerant.)

3 Model of Computation

The model we consider for our problem consists of a collection of processes, channels, and services, which we define formally below. For the rest of this section, we fix:

- I, K, finite index sets, and
- $\mathcal{T}$, a variable type for the entire system, representing the problem being solved, and

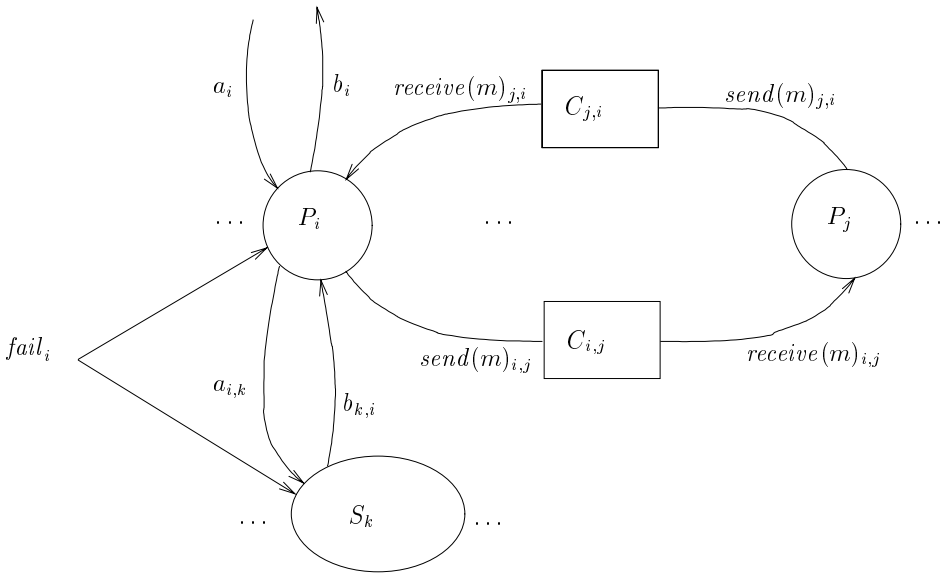

Figure 2: The interfaces of process P_i, channels $C_{i,j}, C_{j,i}$ and service S_k in the complete system.

- M, a message alphabet.

A *distributed system with services* (DSS) for $I, K, \mathcal{T}, M$ is the parallel composition of I/O automata (see [Lyn96, chapter 8]) of the following kinds:

1. *processes* P_i, $i \in I$, and
2. *channels* $C_{i,j}$, $i, j \in I$, $i \neq j$, and
3. *services* S_k, $k \in K$. We let $\mathcal{T}_k$ denote the variable type and $J_k \subseteq I$ denote the set of endpoints of service S_k.

Processes interact only via channels: Process P_i communicates with process P_j over unidirectional channel $C_{i,j}$. Processes also interact with services: Process P_i can invoke service S_k provided that i is in S_k's set of endpoints. Services do not communicate directly with one another; however, they interact indirectly via common processes. Figure 2 shows the interfaces that a process, channel, and service have. In the remainder of this section, we provide more details about the components.

3.1 Processes

Process P_i, $i \in I$ has the following kinds of inputs and outputs:

1. Inputs a_i and outputs b_i, where a is an invocation of type $\mathcal{T}$ and b is a response of type $\mathcal{T}$. These represent P_i's interactions with its own *clients* (the outside world).
2. Outputs $send(m)_{i,j}$ and inputs $receive(m)_{j,i}$, $m \in M$, which connect to channels $C_{i,j}$ and $C_{j,i}$, respectively.
3. For every service S_k such that $i \in J_k$, outputs $a_{i,k}$, where a is an invocation of type $\mathcal{T}_k$, and inputs $b_{k,i}$, where b is a response of type $\mathcal{T}_k$.

4. Input $fail_i$.

We assume that P_i observes well-formedness for each separate service S_k: it does not issue two invocations on S_k without receiving a response to the first one. However, P_i is allowed to issue an invocation on a service without waiting for previous invocations on other services to respond. That is, P_i can issue concurrent invocations to different services, but not to the same service. We also assume that the client of P_i is well-formed with respect to P_i: it does not issue two invocations to P_i without receiving a response to the first one. 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 sends P_i into some kind of state from which (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 fault-tolerant atomic objects defined earlier.

3.2 Services

We define a f-fault-tolerant service of a particular variable type $\mathcal{T}_k$ for a particular set J_k of endpoints, to be simply the canonical f-fault-tolerant atomic object of type $\mathcal{T}_k$ for J_k. Let $\mathcal{T}_k.invs$, $\mathcal{T}_k.resps$ denote the set of invocations, responses, respectively, of the variable type $\mathcal{T}_k$.

The safety properties of a service S_k are determined by its finite traces, which are determined by its start states, transitions, and signature. These are all part of the definition of the service as an I/O automaton. Likewise, the liveness properties of a service S_k are determined by the automaton task structure and the usual conventions for fair executions of I/O automata.

We say that P_i has an *outstanding invocation* to a service S_k iff either (1) the invocation buffer of S_k contains an invocation of the form (i, a), $a \in \mathcal{T}_k.invs$, or (2) the response buffer of S_k contains a response of the form (i, b), $b \in \mathcal{T}_k.resps$.

We say that a service S_k is *silent* along an execution α iff the only actions that S_k executes along α are *dummy* actions.

3.3 Channels

Channel $C_{i,j}$ is a FIFO reliable channel, as defined in [Lyn96, chapter 14]. Its inputs are $send(m)_{i,j}$ actions, which are outputs of P_i, and its outputs are $receive(m)_{i,j}$ actions, which are inputs of P_j. A channel has exactly one task, consisting of its locally controlled actions.

3.4 The task structure of a complete system

The ordinary assumptions about I/O automata mean that the system executes using a "weakly fair" scheduling discipline: in any execution, every task that is continuously enabled gets selected for execution infinitely often. (Thus, an enabled task is eventually either disabled or executed.) For a service S_k, there is a task for each $i \in J_k$, consisting of the actions $\{perform((a, v), (b, v'))_{k,i} : \delta((a, v), (b, v'))\} \cup \{b_i : b \in resps\} \cup \{dummy_{k,i}\}$, see Figure 1. For a process P_i there is a single task, consisting of all the locally controlled actions of P_i. Likewise, for a channel $C_{i,j}$, there is a single task, consisting of all the locally controlled actions of $C_{i,j}$, i.e., the $receive(m)_{i,j}$ actions, $m \in M$.

Since a task of a component contains only its locally controlled actions, we infer from the signature compatibility condition for I/O automata that the tasks define a partition of the set of all actions in the system, except the $init(v)_i$ and $fail_i$ actions; each action occurs in exactly one task.

With this task structure, the weak fairness discipline implies that every message that is sent is eventually received, every process executes infinitely often along an infinite fair execution, and every outstanding invocation (of a service) eventually receives a response.

We introduce a naming scheme for tasks as follows. The single task of P_i, $i \in I$ is called pt_i. The single task of channel $C_{i,j}$, $i, j \in I$, $i \neq j$, is called $ct_{i,j}$. The task of service S_k, $k \in K$ for $i \in J_k$ is called $st_{k,i}$. We define $PT = \{pt_i : i \in I\}$, $CT = \{ct_{i,j} : i, j \in I, i \neq j\}$, $ST = \{st_{k,i} : k \in K, i \in J_k\}$, and $T = PT \cup CT \cup ST$. We call the tasks pt_i $(i \in I)$ *process tasks*, the tasks $ct_{i,j}$ $(i, j \in I, i \neq j)$ *channel tasks*, and the tasks $st_{k,i}$ $(k \in K, i \in J_k)$ *service tasks*.

For any action a except an $init(v)_i$ or $fail_i$, we define $task(a)$ to be the unique t such that $t \in T$ and $a \in t$, i.e., $task(a)$ is the name of the task containing a. We define $task(init(v)_i) = init(v)_i$, and $task(fail_i) = fail_i$, i.e., we consider these actions as being the sole members of singleton tasks, and overload the name of the action as the name of the corresponding task. If e is a channel task $ct_{i,j}$, then let $receiver(e)$ be the process P_j.

3.5 The Consensus problem

The "traditional" specification of f-fault-tolerant consensus is given in terms of a set $\{P_i, i \in I\}$ (I is an index set) of processes that each starts with some value v_i drawn from $\{0, 1\}$. Processes are subject to crash failures [Sch90], that disable the process from producing any output.[1] 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 three conditions [Lyn96, chapter 6]:

Agreement No two processes decide on different values.

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

Termination In every infinite fair execution, all nonfaulty processes eventually decide.

We specify the consensus problem in a slightly different way. We say that a DSS S *solves f-fault-tolerant consensus for I* if and only if S is an f-fault-tolerant atomic object of type *consensus* (Section 2.2) for endpoint set I.

We now show that any system that meets our definition also meets the traditional one. We argue that the f-fault-tolerant canonical consensus object for endpoint set I satisfies the three conditions above (with a slight variation of the termination condition).

From the definition of the *consensus* variable type, each process in I has two invocations, $init(0)$, $init(1)$ and two responses, $decide(0)$, $decide(1)$. By inspecting the consensus variable type given in Section 2.2, we see that the value of the variable is initially $\emptyset$, and on invocation $init(0)$ can change from $\emptyset$ to $\{0\}$, and on invocation $init(1)$ can change from $\emptyset$ to $\{1\}$, and is stable once it is different from $\emptyset$. It is also clear that any $decide(0)$ response is only issued by the object when the variable

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

has value $\{0\}$, and any $decide(1)$ response is only issued by the object when the variable has value $\{1\}$. Hence, after the first $decide(0)$ response, all subsequent responses will be $decide(0)$, and after the first $decide(1)$ response, all subsequent responses will be $decide(1)$. So, the canonical consensus object satisfies the agreement condition. If all invocations are $init(0)$, then the only possible change of the variable is from $\emptyset$ to $\{0\}$. Hence, all responses will be $decide(0)$. Likewise if all invocations are $init(1)$, then all responses will be $decide(1)$. Otherwise, there are both $init(0)$ and $init(1)$ invocations. Hence, in all cases, the value decided on is the value occurring in some invocation. Hence, the canonical consensus object satisfies the validity condition. If at least one process invokes the f-fault-tolerant canonical consensus object, then the value of the variable will eventually be either $\{0\}$ or $\{1\}$, provided that less than f processes fail, and that the scheduling is weakly fair, as discussed in Section 3.4. Hence, all nonfaulty processes that invoke the object will receive a *decide* response, along fair executions in which no more than f processes fail. Processes that do not invoke the object will not receive a response, even if they are nonfaulty. That is, processes that do not invoke the object (with an $init(v)$ action) do not participate in the consensus algorithm, and hence are not required to have an initial value. This is a slightly different condition than the traditional termination condition, which requires that all nonfaulty processes do have an initial value, and that they all eventually decide. Here, only the nonfaulty processes that "participate," by invoking the object, will receive a decision.

Since any system S that solves solves f-fault-tolerant consensus for I can only exhibit behaviors (in composition with a well-formed environment) that are a subset of the behaviours of the f-fault-tolerant canonical consensus object, the desired conclusion follows.

4 The Impossibility Result

The problem we address is to design a system, as given in Section 3, which is an f-fault-tolerant atomic object (Section 2.4) of type *consensus* for some (arbitrary) set I of endpoints. We show that, when the services in the system are restricted to be $(f-1)$-fault-tolerant atomic objects, that this problem is impossible to solve. The services can have arbitrary types, and can have as endpoints any subset of I. Thus, techniques based on quorums, replication, and redundancy, could all be implemented within our model. Our result implies that none of these approaches would help: a limitation on the fault-tolerance of the underlying services is also a fundamental limitation on the fault-tolerance of any consensus service that can be built from these underlying services.

Since we now restrict attention to systems that are consensus objects, the inputs a_i and outputs b_i that represent P_i's interactions with its own clients are now instantiated as the inputs $init(0)_i$, $init(1)_i$, and the outputs $decide(0)_i$, $decide(1)_i$, for the single consensus client that P_i now interacts with.

4.1 Main result and proof assumptions

The main result of the paper is:

Theorem 1 *Let I be an arbitrary endpoint set such that $|I| \geq 2$, and let f be such that $1 \leq f < |I|$. Then there does not exist a distributed system with services that is an f-fault-tolerant atomic consensus object for endpoint set I, if the services are $(f-1)$-fault-tolerant.*

Note that the services can be of any variable type. We assume in the sequel, that such a DSS, P, exists and derive a contradiction.

We assume that all the processes of P are deterministic automata, as defined in Section 2.1. Since channels are FIFO, they are already deterministic. We assume a slightly weaker condition for services, namely that variable type of each service is deterministic, i.e, the relation δ of the underlying variable type is a mapping. For an impossibility proof, these assumptions are made without loss of generality, since processes and services can be made to satisfy the above conditions by removing a subset of the locally-controlled transitions. Hence, if an unrestricted solution exists, then a solution satisfying our assumptions also exists.

4.2 Terminology used in the proof

4.2.1 Transitions

A *transition* is a triple (s, a, s'). We define $first(s, a, s') = s$, $action(s, a, s') = a$, $last(s, a, s') = s'$. The *participants of a locally controlled action* (i.e., not an $init(v)_i$ or $fail_i$ action) a of the system are all automata with a in their signature: $participants(a) = \{A \mid a \in acts(A)\}$. The *participants of a transition* (s, a, s') are the participants of its action: $participants(s, a, s') = participants(a)$.

If the action a of a transition is an output action of some component A (process or service, since channels do not have internal actions), then we say that the transition is an *output transition of* A. We define *internal transition of* A similarly. Due to I/O automaton signature compatibility, a transition can be the output or internal transition of at most one component. Furthermore, due to the structure of the system, as given in Section 3, every transition, with the exception of transitions due to the execution of the $init(v)_i$ inputs to P_i, and $fail_i$ actions, is either an output transition or an internal transition of exactly one component.

4.2.2 Tasks and scheduling

We say that a task e is *applicable* to a global state s iff some action of e is enabled in state s. If α is a finite execution, then we say that e is *applicable* to α iff e is applicable to $last(\alpha)$. Thus, if e is an applicable channel task $ct_{i,j}$, then the corresponding channel $C_{i,j}$ must be nonempty, so that a message can actually be delivered. If e is an applicable service task $st_{k,i}$, then either the invocation buffer of service S_k must contain an invocation from process P_i, or the response buffer of S_k must contain a response to P_i, or the $dummy_{k,i}$ action must be enabled. We assume, for technical convenience, that a process always has an enabled locally controlled action, and so a process task is always applicable.

An applicable task e, together with the current global state, determines a unique transition (arising from the scheduling of task e in the current state) since processes and channels are deterministic, and the variable type underlying a service is also deterministic. We denote this transition as $transition(e, s)$. Let $transition(e, s) = (s, a, s')$. Then, we apply the notation defined in Section 4.2.1 to $transition(e, s)$ as follows: $first(e, s) = s$, $action(e, s) = a$, $last(e, s) = s'$. We abbreviate $last(e, s)$ by $e(s)$. We note that $transition(e, s)$, $first(e, s)$, $action(e, s)$, $last(e, s)$ are defined if and only if e is applicable to s.

We note that when e is a channel task, then $transition(e, s)$ always causes a change of state, i.e., $e(s) \neq s$, since some message is delivered by the channel. When e is a service task $st_{k,i}$, then $transition(e, s)$ causes a change of state unless it corresponds to the execution of a $dummy_{k,i}$ action.

When e is a process task, then $transition(e, s)$ may or may not cause a state change. This would depend on the transition structure of the process, about which we make no assumptions.

4.2.3 Executions

Define an *initialization* of P to be a finite execution containing exactly $|I|$ actions, which moreover are all $init(v_i)_i$ actions, one for each $i \in I$. Define an execution α of P to be *input-first* iff it has an initialization as a prefix, and otherwise contains no *init* actions. If α is a finite execution, then an extension of α is an execution α' such that α is a prefix of α'. Define a finite input-first failure-free execution α to be *0-valent* if (1) some input-first failure-free extension of α contains a $decide(0)_i$ action, for at least one $i \in I$, and (2) no input-first failure-free extension of α contains a $decide(1)_i$ action, for any $i \in I$. The definition of *1-valent* is analogous. Define a finite failure-free execution α to be *univalent* iff it is either 0-valent or 1-valent. Define a finite input-first failure-free execution α to be *bivalent* iff it has some input-first failure-free extension that contains a $decide(0)_i$ action, for at least one $i \in I$, and some input-first failure-free extension that contains a $decide(1)_i$ action, for at least one $i \in I$.

Since the assumed f-fault-tolerant atomic consensus object P is an I/O automaton, we can view its transition relation as defining a labeled directed graph whose nodes are the states of P and which contains a directed edge from s to s' labeled with a iff (s, a, s') is in the transition relation of P. This graph is called the *global state transition graph* of P. Let $G(P)$ be the subgraph of the global state transition graph of P obtained as follows: (1) include every state that lies along an input-first execution, and (2) include all the transitions of P that connect the states that are included by virtue of (1).

4.2.4 Schedules

A *schedule* is a finite sequence of task names drawn from $T \cup \{init(v)_i, fail_i : v \in \{0, 1\}, i \in I\}$. Let $\sigma = e_1 e_2 \ldots e_n$ be a schedule, and s be a global state, such that, e_1 is applicable to s, e_2 is applicable to $e_1(s)$, and, generally, e_i is applicable to $e_{i-1}(e_{i-2}(\ldots(e_1(s))\ldots))$ for all i, $1 < i \leq n$. Then, we say that σ is applicable to s, and we let $\sigma(s)$ denote $e_n(e_{n-1}(\ldots(e_1(s))\ldots))$. A schedule σ is applicable to a finite execution α iff σ is applicable to $last(\alpha)$. In this case, we let $\sigma(\alpha)$ denote the resulting extension of α.

Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots s_{i-1} a_i s_i$ be a finite execution. Then, we define the schedule $schedule(\alpha) = task(a_1) task(a_2) \ldots task(a_i)$. That is, for each action in α, we take the name of the task containing the action. $schedule(\alpha)$ then consists of these task names in the same order as their corresponding actions.

4.3 The proof

Our proof will build up a series of lemmas establishing certain constraints on $G(P)$. We start with the basic commutativity situation illustrated in Figure 3.

Lemma 2 *Let s be any global state of the f-fault-tolerant atomic consensus object P, and let e_1, e_2 be tasks such that*

1. *e_1, e_2 are both applicable to s, and*

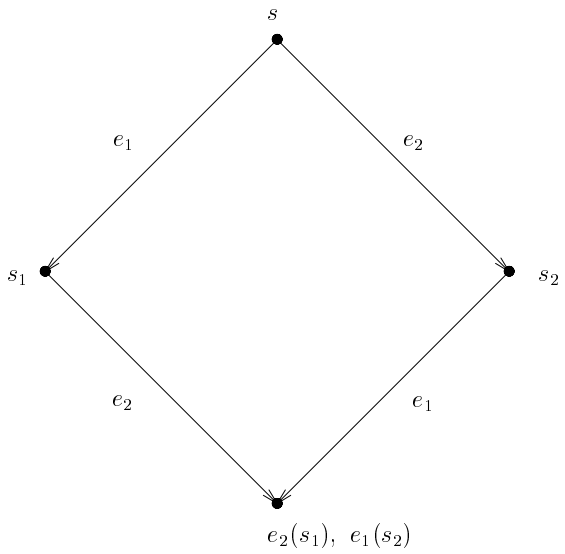

Figure 3: Commuting tasks w.r.t. a state s.

2. $\mathit{participants}(e_1, s) \cap \mathit{participants}(e_2, s) = \emptyset$.

Let $e_1(s) = s_1$, and $e_2(s) = s_2$. Then, e_2 is applicable to s_1, and e_1 is applicable to s_2, and $e_2(s_1) = e_1(s_2)$.

Proof. By assumption (e_1, s) and (e_2, s) only affect the state of different components. It follows that e_2 is applicable to s_1, and that e_1 is applicable to s_2. By determinism, it follows that $\mathit{participants}(e_1, s) = \mathit{participants}(e_1, s_2)$, and that (e_1, s) and (e_1, s_2) are the same transition "locally," i.e, they effect exactly the same state changes in the components in $\mathit{participants}(e_1, s)$. Likewise for (e_2, s) and (e_2, s_1). Thus, the accumulated state changes of (e_1, s) followed by (e_2, s_1) are the same as the accumulated state changes of (e_2, s) followed by (e_1, s_2). Hence the lemma holds. Figure 3 illustrates the proof. □

Lemma 3 *The f-fault-tolerant atomic consensus object P must have a bivalent initialization.*

Proof. Recall that we assume $f \geq 1$ (Section 4.1). The argument is then exactly the same as that in the proof of Lemma 12.3 in [Lyn96, chapter 12]. □

Suppose there exists a finite input-first failure-free execution α_s, and states s, s', s'', s_0, s_1, and tasks e, e' which are related as given by Figure 4. We call such a configuration a *hook*, after [CHT96]. We say that the hook *starts in state s*, and we call α_s the *stem* of the hook. We also admit as a hook a configuration in which the 0-valent and 1-valent states are interchanged.

Lemma 4 *Let α_s be a finite input-first failure-free bivalent execution of $G(P)$, and let $\mathit{first}(\alpha_s) = s_{start}$, $\mathit{last}(\alpha_s) = s$. Let e be a task of P applicable to α_s. Let*

$U = \{\alpha_u \mid \alpha_u = \sigma(\alpha_s),\ \sigma$ is a finite failure-free schedule applicable to α_s and not containing $e\}$,

$V = \{e(\alpha_u) \mid \alpha_u \in U$ and e is applicable to $\alpha_u\}$.

Then either (1) V contains a bivalent execution, or (2) $G(P)$ contains a subgraph which is a hook starting in s_{start}, as given by Figure 4.

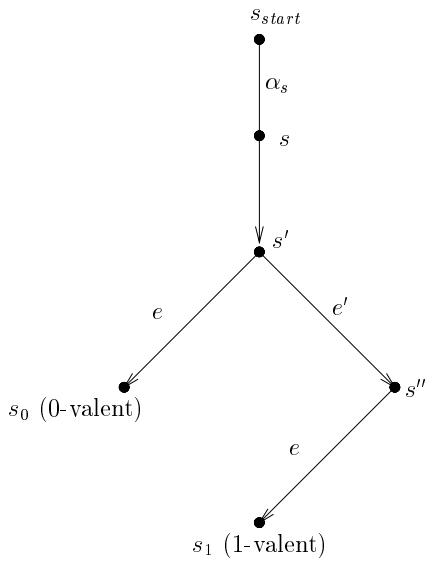

Figure 4: A hook starting in s.

Proof. We assume both the antecedent of the lemma and the negation of (1), and establish (2).

Now e is either a channel task, process task, or service task. If e is a channel task $ct_{i,j}$, then applicability of e to s means that channel $C_{i,j}$ contains a message in state s. Thus, e is also applicable to any state reached from s by a schedule not containing e, since the message remains in $C_{i,j}$ as long as $ct_{i,j}$ is not scheduled. If e is a process task, then e is applicable to any state, by our assumption that a process always has some enabled locally controlled action. If e is a service task $st_{k,i}$, then applicability of e to s means that either service S_k has a pending invocation from process P_i in state s, or $dummy_{k,i}$ is enabled. Thus, e is also applicable to any state reached from s by a schedule not containing e, since the invocation (if present) remains pending as long as $st_{k,i}$ is not scheduled, and $dummy_{k,i}$ remains enabled once it is enabled. We have therefore shown,

$$e \text{ is applicable to every execution in } U. \qquad \text{(a)}$$

Since α_s is bivalent, there exists a 0-valent extension α_{x_0} of α_s and a 1-valent extension α_{x_1} of α_s. For $i \in \{0, 1\}$, we argue as follows.

CASE 1: $\alpha_{x_i} \in U$. Let $\alpha_{v_i} = e(\alpha_{x_i})$. Hence α_{v_i} is i-valent, since α_{x_i} is i-valent. Also, $\alpha_{v_i} \in V$, since $\alpha_{x_i} \in U$.

CASE 2: $\alpha_{x_i} \notin U$. Then, e was applied in extending α_s to α_{x_i}. Let α_{v_i} be the unique extension of α_s whose last action has task e. α_{v_i} is unique due to our assumptions in Section 4.1 about the deterministic behavior of processes and variable types. Hence $\alpha_{v_i} = e(\alpha'_s)$ for some extension α'_s of α_s. Hence $\alpha_{v_i} \in V$ by definition of V. Since (1) is false by assumption, V contains no bivalent executions. Hence α_{v_i} is univalent. But α_{x_i} is i-valent and is an extension of α_{v_i}. Hence α_{v_i} is i-valent.

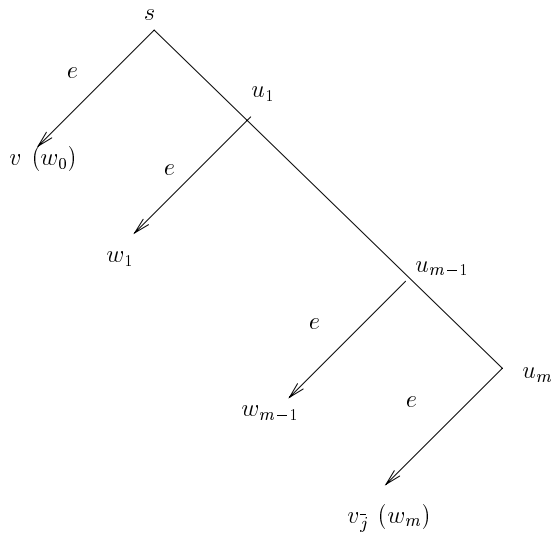

Figure 5: Existence of the hook.

Thus, in both cases, we have that $\alpha_{v_i} \in V$ and α_{v_i} is i-valent. Moreover, this holds for both $i = 0$ and $i = 1$. Thus

$$\text{there exist 0-valent } \alpha_{v_0} \in V \text{ and 1-valent } \alpha_{v_1} \in V \tag{b}$$

Let $\alpha_v = e(\alpha_s)$, and let $v = last(\alpha_v)$. Hence $\alpha_v \in V$, and so α_v is univalent by the assumption that (1) is false. Without loss of generality, let α_v be 0-valent. By (b), there exists $\alpha_{v_1} \in V$ which is 1-valent. Let α_{u_m} be an execution in U such that $e(u_m) = \alpha_{v_1}$, and let $u_m = last(\alpha_{u_m})$. Hence, we have the situation depicted in Figure 5, since α_{u_m} is an extension of α_s. (The state s is the same state in Figures 4 and 5). Consider the (unique) execution fragment γ such that $\alpha_{u_m} = \alpha_s \frown \gamma$. By (a), e is applicable to every state along γ. Since the resulting executions are all in V by definition, they are all univalent, by assumption. Since α_v is 0-valent and α_{v_1} is 1-valent, it follows that there exist two such executions, α_0 and α_1 such that α_0 is 0-valent, α_1 is 1-valent, and α_0, α_1 result from applying e to adjacent states along γ. The subgraph of $G(P)$ generated by taking the "union" of α_0 and α_1 (i.e., take all states and transitions occuring in one, or both, of α_0, α_1) is then the desired hook. □

Lemma 5 *$G(P)$ does not contain as a subgraph a hook whose stem is a finite input-first failure-free execution.*

Proof. Our proof is by contradiction. We assume that $G(P)$ does contain such a hook, and establish that P is not a f-fault-tolerant atomic consensus object, contrary to assumption.

Without loss of generality, we assume the configuration in Figure 4. For each state except s_{start}, we let α subscripted with the state name denote the unique finite execution which is contained in the hook and which ends in that state: $\alpha_{s'}$ is the stem of the hook, α_{s_0} ends in s_0, α_{s_1} ends in s_1, and $\alpha_{s''}$ ends in s''.

We remark that $\alpha_{s'}$ cannot contain any *decide* actions, since it is bivalent, and this would otherwise violate the agreement property. We first establish Claims 1–3.

Claim 1: $e \neq e'$.
Suppose not. Then, by determinism (Section 4.1), we have $s_0 = s''$. Now s_1 is reachable from s'', and s_1 is 1-valent. Hence, s'' is either bivalent or 1-valent. s_0 however, is 0-valent. Hence we have a contradiction. So, claim 1 is established.

Claim 2: $|participants(e, s')| \leq 2$, $|participants(e', s')| \leq 2$.
From the structure of a DSS (Section 3), we see that every output action of some component is an input action of at most one other component. The claim follows.

Claim 3: $|participants(e, s') \cap participants(e', s')| \leq 1$.
From Claim 2, we immediately have that $|participants(e, s') \cap participants(e', s')| \leq 2$. Suppose $|participants(e, s') \cap participants(e', s')| = 2$. From Claim 1, we know that $e \neq e'$. Hence, it must be that, for some distinct components C_1, C_2, $action(e, s')$ is an output action of C_1 and an input action of C_2, $action(e', s')$ is an input action of C_1 and an output action of C_2. Since services and channels have no actions in common, the only possibilities for this are:

- $\{C_1, C_2\} = \{P_i, S_k\}$ for some P_i, S_k.
  This violates well-formedness of P_i for S_k.
- $\{C_1, C_2\} = \{P_i, C_{i,j}\}$ for some P_i, $C_{i,j}$.
  No output action of $C_{i,j}$ is an input action of P_i.
- $\{C_1, C_2\} = \{P_i, C_{j,i}\}$ for some P_i, $C_{j,i}$.
  No output action of P_i is an input action of $C_{j,i}$.

Since all three cases lead to a contradiction, the claim is established.

From Claim 3, we have four possibilities for $participants(e, s') \cap participants(e', s')$. To complete the proof of the lemma, we consider each separately.

CASE 1: $participants(e, s') \cap participants(e', s') = \emptyset$. Hence, the antecedent of Lemma 2 holds for $s = s'$, $e_1 = e$, and $e_2 = e'$. Hence, e' is applicable to s_0, and $e'(s_0) = s_1$. Hence, $e'(\alpha_{s_0})$ and α_{s_1} have at least one infinite fair extension with a common suffix. Since $\alpha_{s'}$ does not contain any *decide* actions, it follows that the suffix must contain *decide* actions. Now α_{s_0} is 0-valent and α_{s_1} is 1-valent. Hence, no matter what *decide* actions this common suffix contains, it will violate the valencies of at least one of α_{s_0}, α_{s_1}.

CASE 2: $participants(e, s') \cap participants(e', s') = S_k$.

Subcase 2.1: *At least one of* $action(e, s')$, $action(e', s')$ *is not a perform action of* S_k. Hence at least one of these is an invocation or a response. Now invocation and response actions do not change the value of the underlying variable of S_k.

Since both these actions are enabled in s', it follows that the enablement of neither action depends on the prior execution of the other action (this might be the case for certain invocation, perform or perform, response pairs of actions, but not here). Hence, from Figure 1, we see that these actions commute, in that their order can be reveresed and the same final global state wil result. Hence, e' is applicable to s_0, and $e'(s_0) = s_1$. Hence, $e'(\alpha_{s_0})$ and α_{s_1} have at least one

infinite fair extension with a common suffix. Since $\alpha_{s'}$ does not contain any *decide* actions, it follows that the suffix must contain *decide* actions. Now α_{s_0} is 0-valent and α_{s_1} is 1-valent. Hence, no matter what *decide* actions this common suffix contains, it will violate the valencies of at least one of α_{s_0}, α_{s_1}.

Subcase 2.2: *Both of* $action(e, s')$, $action(e', s')$ *are perform actions of* S_k. Since $\alpha_{s'}$ is bivalent, then, under the assumption that P solves f-fault-tolerant consensus, $\alpha_{s'}$ cannot contain any *decide* actions, since that would violate agreement. Hence, α_{s_0} does not contain any *decide* actions either, since $action(e, s')$ is not a *decide*.

Let α'' be an infinite fair execution that extends α_{s_0}, and let α' be the suffix of α' starting in state s'. Furthermore, let α' be chosen such that:

1. The first f actions along α' are $fail_j$ actions for f different $j \in J_k$
2. For every ocurrence of an action a along α', and every $i \in I$, if $task(a) = st_{k,i}$, then $a = dummy_{k,i}$. That is, whenever $st_{k,i}$ is scheduled along α, the $dummy_{k,i}$ action is chosen. Since $dummy_{k,i}$ is enabled at all states of α' except the first, it is certainly possible to always choose to schedule the $dummy_{k,i}$ action in this way, along α'.

Since P is f-tolerant, $f \geq 1$, a $decide(v)_j$ action, for every nonfaulty process must occur along α'. Let α'_d be the prefix of α' ending in the state just after the first such $decide(v)_j$ action. Let $\sigma = schedule(\alpha'_d)$. From σ, derive the schedule σ' by removing:

1. Every occurrence of a $fail_i$, and
2. Every occurrence of $st_{k,i}$ for all $i \in I$ (these all correspond to $dummy_{k,i}$ actions in α'_d),

It is clear that σ' is a failure-free schedule. Since, in α, the transitions corresponding to the above task occurrences do not induce any change of state other than to S_k, which is silent, it follows that σ' is applicable to s_0, and that $\sigma'(\alpha_{s_0})$ contains a single *decide* action.

By the case condition, s_0 and s_1 differ only in the state of S_k. Since processes and channels are deterministic, and since services have a deterministic type and also behave as given by Figure 1, we can see that σ' is applicable to s_1, and that $\sigma'(\alpha_{s_1})$ is the same as $\sigma'(\alpha_{s_0})$, with the exception of the local state of S_k. In particular, $\sigma'(\alpha_{s_1})$ and $\sigma'(\alpha_{s_0})$ contain the same action subsequence. So, $\sigma'(\alpha_{s_1})$ and $\sigma'(\alpha_{s_0})$ contain the same single $decide(v)_i$ action, for some $v \in \{0, 1\}$, Choosing $v = 0$ contradicts the 1-valency of s_1, and choosing $v = 1$ contradicts the 0-valency of s_0.

CASE 3: $participants(e, s') \cap participants(e', s') = C_{i,j}$. Since P_i and $C_{i,j}$ are deterministic, and $e \neq e'$, it follows that one of $action(e, s')$, $action(e', s')$, is a $send(m)_{i,j}$, and the other is a $receive(m')_{i,j}$, for some $m, m' \in M$. Since these are both enabled in s', it follows from the definition of a FIFO channel (see [Lyn96, chapter 14]) that $transition(e, s')$ and $transition(e', s')$ commute. The remainder of the argument is similar to Case 2.1.

CASE 4: $participants(e, s') \cap participants(e', s') = P_i$.
Since $\alpha_{s'}$ is bivalent, then, under the assumption that P solves f-fault-tolerant consensus, $\alpha_{s'}$ cannot contain any $decide()$ actions, since that would violate agreement.

Let α'' be an infinite fair execution that extends $\alpha_{s'}$, and let α' be the suffix of α' starting in state s'. Furthermore, let α' be chosen such that:

1. The action along α' that starts in s' is $fail_i$, and
2. No $fail_j$ actions, $j \neq i$, occur along α', and
3. For every action a, and every occurrence of a along α', if $task(a) = st_{k,i}$ for some $k \in K$, then $a = dummy_{k,i}$. That is, whenever $st_{k,i}$ is scheduled along α, the $dummy_{k,i}$ action is chosen. Since $dummy_{k,i}$ is enabled at all states, except the first, of any execution fragment that starts with $fail_i$, it is certainly possible to always choose to schedule the $dummy_{k,i}$ action in this way, along α'.

Since P is f-tolerant, $f \geq 1$, a $decide(v)_j$ action, for every $j \neq i$ must occur along α'. Let α'_d be the prefix of α' ending in the state just after the first such $decide(v)_j$ action. Let $\sigma = schedule(\alpha'_d)$. From σ, derive the schedule σ' by removing:

1. The single occurrence of $fail_i$, and
2. Every occurrence of $st_{k,i}$ for all $k \in K$, (these all correspond to $dummy_{k,i}$ actions in α'_d), and
3. Every occurrence of $ct_{j,i}$, for all $j \in I$, $j \neq i$

Since the only *fail* action along σ is $fail_i$, it is clear that σ' is a failure-free schedule. Since, in α, the transitions corresponding to the above task occurrences do not induce any change of state other than to P_i, which has failed, it follows that σ' is applicable to s', and that $\sigma'(\alpha_{s'})$ contains a single *decide* action. We now establish Claims 4.1 and 4.2.

Claim 4.1:

1. σ' is applicable to s_0.
2. Let γ be the suffix of $\sigma'(\alpha_{s'})$ starting in s', and let γ_0 be the suffix of $\sigma'(\alpha_{s_0})$ starting in s_0. Then γ, γ_0 contain the same decide actions.

We establish the claim by case analysis on the possibilities for $action(e, s')$. From the case 4 condition, we have that $P_i \in participants(e, s')$. This restricts the possibilities for $action(e, s')$ to the following.

Subcase 4.1.1: $action(e, s') = a_{i,k},\ a \in \mathcal{T}_k.invs$. By definition, σ' contains no occurrence of $st_{k,i}$. Hence, γ contains no action in $st_{k,i}$. Let γ_{00} be the same as γ except that, for corresponding states along γ_{00}, the invocation buffer of S_k contains additionally the invocation (i, a). Since γ_{00} contains no action in $st_{k,i}$, this extra invocation is never processed (by a $perform()$ action) along γ_{00}. Hence, the state-action-state triples along γ_{00} are actual transitions of P (i.e., elements of $steps(P)$). Thus, γ_{00} is an actual execution fragment of $G(P)$. Furthermore, the first state of γ_{00} is s_0, and $schedule(\gamma_{00}) = \sigma'$. Hence σ' is applicable to s_0. Now γ_{00} is the suffix of $\sigma'(s_0)$ starting in s_0. Also, γ and γ_{00} contain the same subsequence of actions, and so in particular contain the same *decide* actions. Letting $\gamma_0 = \gamma_{00}$ establishes the claim in this case.

Subcase 4.1.2: $action(e, s') = b_{k,i},\ b \in \mathcal{T}_k.resps$. By definition, σ' contains no occurrence of pt_i nor of $st_{k,i}$. Let γ be the suffix of $\sigma'(\alpha_{s'})$ starting in s'. Hence, γ contains no action in pt_i nor in

$st_{k,i}$. Let γ_{00} be the same as γ except that, for corresponding states along γ_{00}, the response buffer of S_k is missing the response (i, b), and the state of P_i is the result of executing input action $b_{k,i}$ in state s'.

We now argue that every state-action-state triple along γ_{00} is in $steps(P)$, i.e, is an actual transition of P. Since γ_{00} contains no actions in pt_i, this difference in P_i's local state does not cause any state-action-state triple along γ_{00} to not be a transition of P, since no action along γ_{00} either depends on (for enablement) nor changes P_i's local state. Likewise, since γ_{00} contains no actions in $st_{k,i}$, then the difference in the response buffer of S_k cannot cause any state-action-state triple along γ_{00} to not be a transition of P, since no action along γ_{00} either depends on (for enablement) those elements of S_k's response buffer of the form (i, b), nor does any such action add or remove elements of the form (i, b) to S_k's response buffer. Thus, γ_{00} is an actual execution fragment of $G(P)$. Furthermore, the first state of γ_{00} is s_0, and $schedule(\gamma_{00}) = \sigma'$. Hence σ' is applicable to s_0. Now γ_{00} is the suffix of $\sigma'(s_0)$ starting in s_0. Also, γ and γ_{00} contain the same subsequence of actions, and so in particular contain the same *decide* actions. Letting $\gamma_0 = \gamma_{00}$ establishes the claim in this case.

Subcase 4.1.3: $action(e, s') = send(m)_{i,j},\ m \in M$. By definition, σ' contains no occurrence of pt_i. Let γ be the suffix of $\sigma'(\alpha_{s'})$ starting in s'. Hence, γ contains no action in pt_i. Also, message m is not received by P_j along γ, since it was not sent. (Wlog, we assume that all messages are tagged with unique identifiers. This is for the purpose of the proof only, and is not a restriction on the assumed system P.) Let γ_{00} be the same as γ except that, for corresponding states along γ_{00}, $C_{i,j}$ contains in addition message m at its end (i.e., m is the "last" message in $C_{i,j}$, recall that channels are FIFO), and the state of P_i is the result of executing output action $send(m)_{i,j}$ in state s'.

We now argue that every state-action-state triple along γ_{00} is in $steps(P)$, i.e, is an actual transition of P. Since γ_{00} contains no actions in pt_i, this difference in P_i's local state does not cause any state-action-state triple along γ_{00} to not be a transition of P, since no action along γ_{00} either depends on (for enablement) nor changes P_i's local state. Likewise, the difference in the contents of $C_{i,j}$ cannot cause any state-action-state triple along γ_{00} to not be a transition of P. The only triples that could possibly be affected are those whose action is $receive(m')_{i,j}$ for some $m' \in M$. But all such triples will correspond to the reception of the message m' actually at the head of $C_{i,j}$ (in the initial global state of the triple), since the only difference in the contents of $C_{i,j}$ is that an extra message has been appended at the rear of $C_{i,j}$. In other words, $C_{i,j}$ delivers the same sequence of messages along γ_{00} that it does along γ. Hence, all these triples will be actual transitions of P. Thus, γ_{00} is an actual execution fragment of $G(P)$. Furthermore, the first state of γ_{00} is s_0, and $schedule(\gamma_{00}) = \sigma'$. Hence σ' is applicable to s_0. Now γ_{00} is the suffix of $\sigma'(s_0)$ starting in s_0. Also, γ and γ_{00} contain the same subsequence of actions, and so in particular contain the same *decide* actions. Letting $\gamma_0 = \gamma_{00}$ establishes the claim in this case.

Subcase 4.1.4: $action(e, s') = receive(m)_{j,i},\ m \in M$. By definition, σ' contains no occurrence of pt_i nor of $ct_{j,i}$. Let γ be the suffix of $\sigma'(\alpha_{s'})$ starting in s'. Hence, γ contains no action in pt_i nor in $ct_{j,i}$. Let γ_{00} be the same as γ except that, for corresponding states along γ_{00}, $C_{j,i}$ is missing the message m at its head, and the state of P_i is the result of executing input action $receive(m)_{j,i}$ in state s'.

We now argue that every state-action-state triple along γ_{00} is in $steps(P)$, i.e, is an actual transition of P. Since γ_{00} contains no actions in pt_i, this difference in P_i's local state does not cause any state-action-state triple along γ_{00} to not be a transition of P, since no action along γ_{00} either depends on (for enablement) nor changes P_i's local state. Likewise, the difference in the contents of $C_{j,i}$ cannot cause any state-action-state triple along γ_{00} to not be a transition of P, since γ_{00} contains no action in $ct_{j,i}$. Thus, γ_{00} is an actual execution fragment of $G(P)$. Furthermore, the first state of γ_{00} is s_0, and $schedule(\gamma_{00}) = \sigma'$. Hence σ' is applicable to s_0. Now γ_{00} is the suffix of $\sigma'(s_0)$ starting in s_0. Also, γ and γ_{00} contain the same subsequence of actions, and so in particular contain the same *decide* actions. Letting $\gamma_0 = \gamma_{00}$ establishes the claim in this case.

Subcase 4.1.5: $action(e, s') = decide(v)_i$ *or* $action(e, s')$ *is an internal action of* P_i. By definition, σ' contains no occurrence of pt_i. Let γ be the suffix of $\sigma'(\alpha_{s'})$ starting in s'. Hence, γ contains no action in pt_i. Let γ_{00} be the same as γ except that, for corresponding states along γ_{00}, $C_{j,i}$ and the state of P_i is the result of executing $action(e, s')$.

We now argue that every state-action-state triple along γ_{00} is in $steps(P)$, i.e, is an actual transition of P. Since γ_{00} contains no actions in pt_i, this difference in P_i's local state does not cause any state-action-state triple along γ_{00} to not be a transition of P, since no action along γ_{00} either depends on (for enablement) nor changes P_i's local state. Thus, γ_{00} is an actual execution fragment of $G(P)$. Furthermore, the first state of γ_{00} is s_0, and $schedule(\gamma_{00}) = \sigma'$. Hence σ' is applicable to s_0. Now γ_{00} is the suffix of $\sigma'(s_0)$ starting in s_0. Also, γ and γ_{00} contain the same subsequence of actions, and so in particular contain the same *decide* actions. Letting $\gamma_0 = \gamma_{00}$ establishes the claim in this case.

From our definition of distributed system with services, we see that the above are all the possible cases for $action(e, s')$. Having established Claim 4.1 in each case, we conclude that it holds generally. (end proof of Claim 4.1)

Claim 4.2:

1. σ' is applicable to s_1.
2. Let γ be the suffix of $\sigma'(\alpha_{s'})$ starting in s', and let γ_1 be the suffix of $\sigma'(\alpha_{s_1})$ starting in s_1. Then γ, γ_1 contain the same decide actions.

From the case 4 condition, we have that $P_i \in participants(e', s')$. Hence, we can apply exactly the same argument as used in the proof of Claim 1 to conclude that:

1. σ' is applicable to s''.
2. Let γ'' be the suffix of $\sigma'(\alpha_{s''})$ starting in s''. Then γ, γ'' contain the same decide actions.

From the case 4 condition, we have that $P_i \in participants(e, s')$. Hence, $e = pt_i$, or $e = ct_{j,i}$, or $e = st_{k,i}$, with $action(e, s') = b_{k,i}$ for some $b \in \mathcal{T}_k.resps$. If $e = pt_i$ or $e = ct_{j,i}$, then clearly $P_i \in participants(e, s'')$. If $e = st_{k,i}$, with $action(e, s') = b_{k,i}$ for some $b \in \mathcal{T}_k.resps$, then, by well-formedness of P_i w.r.t. S_k, and $P_i \in participants(e', s')$, it follows that $action(e', s') \neq a_{k,i}$ for all $a \in$. From $e \neq e'$ it follows that $action(e', s') \neq b_{k,i}$ for all $b \in \mathcal{T}_k.resps$, since otherwise we would have $e' = e = st_{k,i}$. Hence, from $P_i \in participants(e', s')$, we conclude $S_k \notin participants(e', s')$.

Hence, the local state of S_k is the same in s' and s'', i.e., $s'|S_k = s''|S_k$. Since $action(e, s') = b_{k,i}$, we know that in state s', (i, b) is in the response buffer of S_k. Hence, we conclude that in state s'', (b, i) is in the response buffer of S_k. Thus, by well-formedness of P_i w.r.t. S_k, in state s'', the invocation buffer of S_k contains no invocation (i, a), for any $a \in \mathcal{T}_k.invs$. Now s'' lies along a fault-free execution. Hence, $dummy_{k,i}$ is not enabled in s''. Hence, in state s'', the only action of task $st_{k,i}$ that is enabled is $b_{k,i}$ (see Figure 1). Hence $action(e, s'') = b_{k,i}$. Hence $P_i \in participants(e, s'')$.

Thus, for all possible cases of e, we have established $P_i \in participants(e, s'')$. Hence, from (1) σ' is applicable to s'', and (2) γ, γ'' contain the same decide actions, which we showed above, we can apply exactly the same argument as used in the proof of Claim 1 to establish Claim 4.2.
(end proof of Claim 4.2)

Since σ' is a failure-free schedule, and α_{s_0} is a finite failure-free execution, we conclude that $\sigma'(\alpha_{s_0})$ is a finite failure-free execution. Since s_0 is 0-valent, it follows that $\sigma'(\alpha_{s_0})$ contains at least one $decide(0)_j$ action, for some $j \in I$.

Since σ' is a failure-free schedule, and α_{s_1} is a finite failure-free execution, we conclude that $\sigma'(\alpha_{s_1})$ is a finite failure-free execution. Since s_1 is 1-valent, it follows that $\sigma'(\alpha_{s_1})$ contains at least one $decide(1)_{j'}$ action, for some $j' \in I$.

Let γ be the suffix of $\sigma'(\alpha_{s'})$, γ_0 be the suffix of $\sigma'(\alpha_{s_0})$, and γ_1 be the suffix of $\sigma'(\alpha_{s_1})$. From Claims 4.1 and 4.2, we have that γ, γ_0, and γ_1 all contain the same *decide* actions. By its construction, γ contains a single decide action. Hence, γ_0, γ_1 contain a single $decide(v)_\ell$ action in common, for some $v \in \{0, 1\}$, $\ell \in I$. Choosing $v = 0$ contradicts the 1-valency of s_1, and choosing $v = 1$ contradicts the 0-valency of s_0. Hence, we have derived the desired contradiction.
(end of CASE 4)

Since we have established a contradiction in all of CASES 1–4, the lemma holds. □

Lemma 6 *Let α_s be a finite input-first failure-free bivalent execution of $G(P)$, and let $last(\alpha_s) = s$. Let e be a task of P applicable to α_s. Let*
$U = \{\alpha_u \mid \alpha_u = \sigma(\alpha_s),\ \sigma$ *is a finite failure-free schedule applicable to α_s and not containing $e\}$,*
$V = \{e(\alpha_u) \mid \alpha_u \in U$ *and e is applicable to $\alpha_u\}$.*
Then V contains a bivalent execution.

Proof. In the statement of Lemma 4, α_s is a finite failure-free execution and σ is a finite failure-free schedule. Hence, condition (2) of Lemma 4 is the existence of a hook in $G(P)$ whose stem is a finite input-first failure-free execution. By Lemma 5, we know that (2) cannot hold. Thus, the desired result follows immediately from Lemma 4. □

We now present the proof of Theorem 1:

Assume that P is such a distributed system with services. Using Lemma 6, we construct an infinite execution γ of P in which no *decide* action occurs. By Lemma 3, P must have a bivalent initialization. Call it γ_0. We now apply Lemma 6 to extend γ_0 repeatedly.

Fix an arbitrary round-robin order of all the tasks in P, except for the $init(v)_i$ and $fail_i$ tasks. Let γ_i be the current execution, and let t_j be the next task in the round robin order. Assume inductively that γ_i is bivalent. (γ_0 gives the base case).

If t_j is not applicable to $last(\gamma_i)$, then move on to the next task in the round robin order, etc. until an applicable task is found. Since the process tasks are always applicable, we are guaranteed to find an applicable task. So, without loss of generality, let t_j be this task.

By Lemma 6, there is a bivalent extension γ_{i+1} of γ_i such that the last action along γ_{i+1} is in task e.

Let γ be the unique execution such that for all $i \geq 0$, γ_i is a prefix of γ. If a task t is continuously enabled, then, when it is selected in the round robin order, it will be found applicable to the last state of the current execution. Hence, the extension will contain an action from t. Along γ, this will happen infinitely often. Hence, γ satisfies the I/O automaton weak fairness condition. Since γ has infinitely many prefixes γ_i, $i \geq 0$, that are executions of P, it thus follows that γ is an execution of P. Since none of the γ_i contain a *decide* action, it follows that γ does not either. □

5 k-set consensus

We now show that when the system is solving a problem that is weaker than consensus, namely k-consensus (section 2.2), it is possible to boost the fault-tolerance level. Assume we have available f-fault-tolerant k-consensus services, each one with m ports. An f'-fault-tolerant algorithm that solves k'-consensus is 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. If and when it gets a decision back, it sends the decision to all the other processes in the entire set of processes (not just those involved in the same consensus service). Meanwhile, each principal process collects all the results it receives from all processes, and decides on any of these results. The remaining processes simply wait for a result from one of the principal processes. 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 prove correctness, we divide the principal processes appropriately into the services they access. We must ensure that less than $s \cdot (f+1)$ principal process can fail, i.e., $f' < s \cdot (f+1)$, to guarantee that at least one service S has at most f failures. Service S is therefore not killed, and moreover, S has at least one nonfaulty participant, who succeeds in sending the value to everyone. That means that every nonfaulty process decides. The value of k', i.e., 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 fault-tolerance f', we want the smallest possible k' and so we find the smallest integer s that guarantees $f' < s \cdot (f+1)$. Thus we use $s = \lceil (f'+1)/(f+1) \rceil$ 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 7 *For any $1 \leq k < m$, $k \leq f \leq m-1$, $1 \leq f' \leq n-1$, it is possible to solve f'-tolerant k'-consensus for an endpoint set of n processes using f-tolerant k-consensus services, each one with m ports, for*

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

When each service is completely reliable, that is $f = m-1$, and we divide the processes as described above, this algorithm reduces to the one of [HR00], and gives an upper bound proved to

be tight using topology. As an example, we want to build an $f' = 2c-1$-fault-tolerant algorithm for an endpoint set containing at least $2c$ processes, and using only 1-fault-tolerant 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 with 2 processes ($f' + 1 = 2c$ principal processes).

6 Further Work and Conclusions

We studied the consensus problem in an asynchronous distributed system with stopping failures, and where processes can access services that abstract oracles such as hardware primitives or failure detectors. Many papers have studied a similar model, but to our knowledge this is the first time services that are implemented by the processes in the system are considered. We showed that f-tolerant consensus is not achievable using less fault-tolerant consensus services as building blocks, but that k-consensus can be solved with less fault-tolerant k'-consensus services as building blocks.

Our algorithm for k-consensus generalizes that of [HR94, HR00] for reliable services. That algorithm achieves a tight upper bound. It is an open question what is the exact situation for k-set consensus in our model: for which k, k', f, f' is it possible to construct a k-consensus service tolerating f failures from k'-consensus services tolerating f' failures each? This seem to lead to more general hierarchy results, in the style of Herlihy's universality result [Her91], the consensus wait-free hierarchy [Jay97], and the set-consensus hierarchy e.g. [BG93], all of these for services that can fail in our sense.

References

[AGMT95] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared objects. *J. ACM*, 42(6):1231–1274, 1995.

[BG93] E. Borowsky and E. Gafni. The implication of the borowsky-gafni simulation on the set-consensus hierarchy. University of California, Los Angeles, Technical Report 930021, 1993.

[BGLR01] Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, July 2001.

[CHT96] T.D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[CJT94] V. Chandra, T.D.and Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *13'th ACM Symposium on the Principles of Distributed Computing (PODC)*, pages 334–343, 1994.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 11(1):124–149, Jan. 1991.

[HR94] Maurice Herlihy and Sergio Rajsbaum. Set consensus using arbitrary objects. In *Thirteenth Annual ACM Symposium on the Principles of Distributed Computing*, pages 324–333, Los Angeles, CA, August 1994.

[HR00] Maurice Herlihy and Sergio Rajsbaum. Algebraic spans. *Mathematical Structures in Computer Science (Special Issue: Geometry and Concurrency)*, 10(4):549–573, August 2000.

[Jay97] P. Jayanti. Robust wait-free hierarchies. *J. ACM*, 44(4):592–614, July 1997.

[JCT98] P. Jayanti, T.D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, May 1998.

[LAA87] M. C. Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Adv. Comput. Res.*, 4:163–183, 1987.

[LH00] W-K. Lo and V. Hadzilacos. On the power of shared object types to implement one-resilient consensus. *Distributed Computing*, 13(4):219–238, 2000.

[Lyn96] N. A. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, San Francisco, California, USA, 1996.

[Sch90] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

A Technical Background

Definition 1 (I/O Automaton) *An* I/O automaton A *consists of five components:*

1. *A set of states* $states(A)$.
2. *A nonempty set* $start(A) \subseteq states(A)$ *of start states.*
3. *A signature* $sig(A) = (in(A), out(A), int(A))$ *where* $in(A)$, $out(A)$, *and* $int(A)$ *are disjoint sets of input, output, and internal actions, respectively. Denote by* $local(A)$ *the set* $out(A) \cup int(A)$ *and by* $acts(A)$ *the set* $in(A) \cup out(A) \cup int(A)$.
4. *A task partition* $tasks(A)$, *which is a partition of* $local(A)$ *into at most a countable number of classes.*
5. *A transition relation* $steps(A) \subseteq states(A) \times acts(A) \times states(A)$

Let $s, s', u, u, , \ldots$ range over states and $a, b, ...$ range over actions. We say that a is *enabled* in state s iff there exists state s' such that $(s, a, s') \in steps(A)$. If t is a task and some action $a \in t$ is enabled in state s, then we say that task t is enabled in state s.

An execution fragment of A is an alternating sequence of states and actions $s_0 a_1 s_1 \ldots s_{i-1} a_i s_i \ldots$ such that for all $i \geq$, $(s_{i-1} a_i s_i) \in steps(A)$, i.e., the sequence conforms to the transition relation of A. An execution of A is an execution fragment that begins with a state in $start(A)$.

If α is a finite execution or execution fragment, then $first(\alpha)$ denotes the first state of α, and $last(\alpha)$ denotes the last state of α. If α is a finite execution or execution fragment, α' is an execution fragment, and $last(\alpha) = first(\alpha')$, then $\alpha^\frown\alpha'$ denotes the concatenation of α and α'.