Revisiting the Impossibility for Boosting Service Resilience

Xingwu Liu[1], Zhiwei Xu[1], and Juhua Pu[2]

[1] Research Center for Grid and Service Computing, Institute of Computing Technology, Chinese Academy of Sciences
[2] School of Computer Science, BeiHang University
xlw@software.ict.ac.cn, zxu@ict.ac.cn, pujh@buaa.edu.cn

Abstract. An asynchronous distributed system consisting of a collection of processes interacting via accessing shared services or variables. Failure-tolerant computability for such systems is an important issue, but too little attention has been paid to the case where the services themselves can fail. Recently, it's proved that consensus problem can't be $(f+1)$-resiliently solved using a finite number of reliable registers and f-resilient services (failure-aware services must be fully connected). We generalize the result in two dimensions. Firstly, it's shown that the impossibility holds even if infinitely many registers and services are allowed. Secondly, we prove that replacing the reliable registers with reliable shared variables still leave the impossibility to hold, if only failure-oblivious services are allowed.

1 Introduction

1.1 Background

Asynchronous distributed systems consist of a finite collection of processes interacting via some shared mechanisms. The examples of such mechanisms include shared variables, communication channels, atomic objects, and services [1, 2]. A service is itself an arbitrary distributed system with more complicated interface than atomic objects. The major difference between atomic objects and services lies in that a response of an atomic object at a port must exclusively correspond to a previous invocation at the same port, while an invocation to a service at a port may lead to an finite number of responses at any ports, and the service itself may generate responses even without invocations.

Fault-tolerant computability problem is fundamental for asynchronous distributed systems, which explores whether a task can be solved by asynchronous distributed systems, when the components may be subject to failures?

In the related massive literature, consensus tasks [3] are most frequently studied, because they are fundamental in the sense that every atomic object can be implemented from consensus objects and shared read/write variables [4]. [2] also deals with the computability of consensus tasks, proving that there is no $(f+1)$-resilient implementation of consensus objects from canonical f-resilient

J.-Y. Cai, S.B. Cooper, and H. Zhu (Eds.): TAMC 2007, LNCS 4484, pp. 715–727, 2007.
© Springer-Verlag Berlin Heidelberg 2007

failure-aware services, canonical f-resilient failure-oblivious services, and canonical reliable registers, if each failure-aware service is required to be connected with each process (the requirement is called full connection). [2] is noticeable because it's the first paper and presently the only one studying the computability using fault-prone services, while most of the related work only considers reliable atomic objects.

Note that the results in [2] are based on two assumptions. **Finiteness assumption**: only a finite number of services and registers are included in a system. **Distribution assumption**: canonical registers, which are atomic objects of *read/write* type, are used instead of shared variables. However, systems not satisfying these assumptions are also of significance in the theory of distributed computing. For example, in the well-known Herlihy's hierarchy [4], the universal implementation of an n-process atomic object needs infinitely many n-consensus objects if the n-consensus objects are not augmented with a *reset* operation, and more seriously, the hierarchy can't be maintained if read/write shared memory is not available [5].

1.2 Problem Addressed

The following question naturally arises: can we implement a $(f+1)$-resilient consensus object from infinitely many reliable read/write variables and f-resilient services?

This question can't be answered as a trivial corollary of the results in [2], since no one has proven that a system with shared variables and fault-prone services can be simulated by one with reliable atomic objects and fault-prone services. [6] shows that any shared variable system can be simulated by a shared atomic object system, but it relies on (1) that a process and its user are exclusively enabled, which fails to hold for our case, and (2) all processes in the simulated system interact only via shared variables, while services are also allowed in our systems.

Furthermore, two obstacles exist if we straightforwardly borrow the proof in [2]. Firstly, the proof of [Lemma 5, 2] doesn't carry over, as shown in Subsection 3.2. Secondly, Claim 3 in [Lemma 8, 2] fails if shared reliable read/write variables are used instead of canonical registers, due to the reason in Subsection 4.1.

1.3 Related Work

There is a long line of research on the computability of distributed decision problems in shared object systems. However, much of it focuses on fault-prone processes and assumes that the shared variables and objects are reliable. Well-known examples include the impossibility of set-consensus/renaming using shared read/write variables [7,8], the solvability of set consensus using arbitrary objects under certain conditions [9], and the Herlihy's hierarchy.

Afek et al. [10] first studied the solvability of consensus with faulty shared memory. It's shown that faults do not qualitatively decrease the power of such

primitives as test-and-set and read-modify-write, in that they retain their positions in Herlihy's hierarchy. The failures of a variable are spontaneous modifications of its value, modeled by arbitrary *writes* from the environment of the whole system.

Though Afek et al. have focused on a few types of shared memory, Jayanti et al. [11] studied implementing arbitrary objects with fault-prone base objects. They classify object failures into responsive and non-responsive, and shows that any type of object can be implemented so as to keep reliable when some base objects are responsively faulty, and that any non-responsive fault can't be tolerated in this sense. The faults considerd by Afek et al belong to the responsive class.

Attie et al [2] go a step further by considering implementing objects using general services rather than only shared variables and atomic objects. Another critical difference of [2] from [10, 11] lies in the failure model. In [10,11], a base object fails spontaneously, i.e. independently of the processes; however, [2] models a failure of a service with an input action *fail* from the system environment. Because any client process of the service shares the *fail* action with the service, the failures of a service and its processes are not independent. The main result of Attie et al is that it's impossible to implement a $(f+1)$-resilient consensus object from canonical reliable registers, f-resilient canonical atomic objects, and f-resilient services. When failure-aware services are allowed, full connectivity should be satisfied in order to maintain the impossibility. The result is also the starting point of the present paper.

1.4 Our Contribution

We generalize [2] in two dimensions. We first discard the finiteness assumption, proving that all the results in [2] hold even if an infinite number of services and canonical registers can be in one system. This is achieved by a novel scheduling approach such that each of an infinite collection of tasks has infinitely many opportunities to be scheduled. Then we discard the distribution assumption, showing that using the seemingly *more synchronized* shared read/write variables instead of canonical registers still maintains the impossibility results if failure-aware services are not allowed. For this end, a generalized version of [Theorem 13.7, 6] is proposed, indicating that any distributed system where processes communicate via reliable shared variables and services can be simulated by another one with the shared variables replaced by reliable atomic objects of the corresponding types. This step is itself very interesting, since it frees us from many constraints in constructing or optimizing an asynchronous distributed system.

1.5 Organization

The rest of this paper is organized as follows. Section 2 presents some preliminaries. Section 3 and Section 4 are devoted to the two dimensions of our generalization, respectively. And Section 5 concludes this paper.

2 Preliminaries

We assume the terminology of [2] and [6]. Some concepts are mentioned below. For more detail, please refer to the references.

2.1 Sequential Types, Atomic Objects, and Services

We remind the notion of a "sequential type", in order to describe allowable sequential behavior of atomic objects. It's borrowed from [2]. A sequential type $\Gamma = \langle\ V,\ V_0$, invs, resps, $\delta\ \rangle$ consists of:

- V, a nonempty set of values,
- $V_0 \subseteq V$, a nonempty set of initial values,
- invs, a set of invocations,
- resps, a set of responses, and
- δ, a relation from invs $\times V$ to 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 $((a,v),\ (b,\ v'))\in\delta$.

We sometimes use dot notation, writing $\Gamma.V$, $\Gamma.V_0$, Γ.invs, . . . for the components of Γ.

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

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

An atomic object of sequential type Γ is an automaton whose behavior, if serialized in some way, satisfies Γ. An atomic object of read/write type is called a register, and that of binary consensus type is called a consensus object. An atomic object is f-resilient if when no more than f ports fail, all non-failed ports can provide correct behavior. A canonical f-resilient atomic object of type Γ is a special atomic object which describes the allowable concurrent behavior of all f-resilient atomic objects of type Γ. For more about atomic objects, see [2, 6].

A service is a generalized notion of atomic object. Services can be classified into failure-oblivious and failure-aware ones, depending on whether they provide a port with the failure information of other ports. The behavior of a service is also specified by a service type. For more about services, see [2].

In the present paper and [2], both atomic objects and services permit concurrent operations at the same or different endpoints, in the sense that multiple invocations can be issued, without waiting for responses to the previous ones.

2.2 Shared Variable Systems with Services

In [2], all the systems considered consist of processes, reliable registers, and failure-prone services, and have only a finite number of components. In Section 4

of the present paper, we will consider a similar system model which is different in two aspects. Firstly, we use shared reliable variables instead of reliable register. Secondly, an infinite number of services and variables may be included in a system. The system in fact is composed of a shared variable subsystem and a collection of services, with the actions used to communicate among the components hidden.

The shared variable subsystem is modeled as a single I/O automaton. The interface of a process P_i includes invocation and response actions to interact with the external world, input and output actions to invoke and receive response from the services, and an input $fail_i$ to model an unexpected failure. We assume that the $fail_i$ input action affects P_i in such a way that, from that point onward, no output actions or shared variable actions are enabled. It's supposed that in any state, a process P_i always has an action enabled.

A process can access the shared variables only by internal actions, and each action can access at most one shared variable. So, we further distinguish between two different kinds of internal actions: those that involve the shared variables (called shared variable actions) and those don't. If a variable x is of sequential type $\Gamma x = \langle V, V_0$, invs, resps, $\delta \rangle$, each shared variable action accessing it by P_i must be of the form:

Precondition: $p(state_i)$ //p is a predicate of P_i's current state
Effect: $(b, x) \leftarrow \delta(a, x)$ // $a \in \Gamma x.invs$
$state_i) \leftarrow$ any s such that $(state_i)$, b, $s)$ belongs to g //g is a certain relation

In the complete system, processes interact only via services and variables, services don't communicate directly with one another, and can't access the shared variables. The interface of the complete system consists of all the invocation and response actions of the processes, plus $fail_i$ for every process P_i.

An issue has to be clarified. Composing I/O automata is usually under the following condition: each action is shared by only a finite number of component automata. Otherwise some properties may fail to hold, for example, [Theorems 8.3 and 8.5, 6]. However, in our specification, two services S_1 and S_2 share the input action $fail_i$ if i is an endpoint for both S_1 and S_2, so it's possible that infinitely many services share an input action. Fortunately, whether this phenomenon occurs is irrelevant to the proof of our first main result, and our second main result only uses services such that necessary properties still hold even if infinitely many services share some $fail_i$.

2.3 The Implementation of Consensus Objects

We also consider boosting resilience in implementing f-resilient consensus objects. By the definition of binary consensus type, an n-process f-resilient consensus objects satisfies:

- Agreement. No two processes decide on different values,
- Validity. Any value decided on is the initial value of some process,

– Termination. In every fair execution in which at most f processes fail, any non-faulty process receives an input eventually decides.

3 Impossibility for Infinite Systems

To begin our argument, the idea of [2] has to be briefly reviewed. In this section, a *finite* (*respectively*, *infinite*) *system* will stand for a system with a finite (respectively, infinite) collection of reliable registers and fault-prone services.

3.1 A Brief Review

All the impossibility results of Theorem 1, 10, and 11 in [2] can be restated collectively as the following proposition.

Proposition 1. (f+1)-resilient consensus objects can't be implemented from finitely many canonical reliable registers and canonical f-resilient services if each failure-aware service must be connected to all the processes.

The proof is by contradiction and includes seven lemmas. Assume that there is a finite system C to solve such a problem. In [Lemma 2, 2], it's shown that once a task of C is enabled in the final state of a finite execution, it's enabled from then on until it's applied. This commutability is used in proving [Lemma 5, 2]. [Lemma 3, 2] is trivial, claiming that every finite failure-free input-first execution of C is either bivalent or univalent, based on which [Lemma 4, 2] guarantees the existence of a finite bivalent execution. Then [Lemma 5, 2] shows that from the finite bivalent execution one can construct a hook-like subgraph of the graph $G(C)$ of C. [Lemmas 6 and 7, 2] prove that two univalent finite executions, if similar in some sense, must have the same valence, which leads to a contradiction that $G(C)$ contains no hooks, as stated in [Lemma 8, 2]. So, such a finite system C doesn't exist.

For convenience and without loss of generality, [2] assumes that the processes P_i, $i \in I$, are deterministic automata in the sense that in each state s, there is at most one transition (s, a, s') such that a is non-input action. The services are also assumed to be deterministic in the sense that its type has a single initial value and the transition relations are mappings. It's also assumed that each process has a single task, and always has an action enabled. In this section, we adopt these assumptions, so any failure-free execution of C can be defined by applying a sequence of tasks, one after the other, to the initial state of C.

The above mentioned $G(C)$ is defined as follows.

(1) The vertices of $G(C)$ are the finite failure-free input-first extensions of the finite bivalent execution α_b).

(2) $G(C)$ contains an edge labeled with task e from α to α' provided that $\alpha' = e(\alpha)$, the extension of α with the task e.

By the determinism assumption, for any vertex α of $G(C)$ and any task e, there is at most one edge labeled with e outgoing from α.

The above mentioned hook is a subgraph of $G(C)$ of the form in Fig. 1, where s_1 and s_2 are univalent but have different valence.

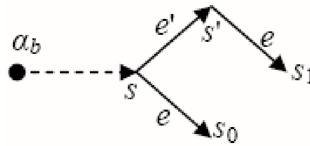

Fig. 1. A hook. (from [2]).

3.2 A Generalization to the Case of Infinite Systems

We prove that the above Proposition 1 also holds even if infinite systems are allowed. Our main result in this section is Theorem 2.

Theorem 2. $(f+1)$-resilient consensus objects can't be implemented from infinitely many canonical reliable registers and canonical f-resilient services if each failure-aware service must be connected to all the processes.

As in [2], we also proceed by contradiction and perform an analysis on the hook structure. So, assume there is an infinite system C to implement $(f+1)$-resilient consensus objects. [Lemmas 2-8, 2], except [Lemma 5, 2], all carry over since they don't care whether the number of canonical reliable registers and canonical f-resilient services is finite or infinite. Thus, if only [Lemma 5, 2] keeps correct, our Theorem 2 holds. However, the proof of [Lemma 5, 2] doesn't carry over, because it extend an execution with an infinite number of tasks in a round-robin fashion. When infinitely many services are used, there are infinitely many tasks, so the round-robin extension must be modified in order to guarantee that the resulting execution, if infinite, is fair. This new version of [Lemma 5, 2] is presented as the following Lemma 3.

In an infinite system, there are infinitely many tasks, so the idea of our modification is to extend the execution in infinite rounds, and to consider only a finite collection of tasks (called candidate task set for this round) for each round. If the candidate task sets are chosen properly, every task eventually has infinite opportunities to be considered.

Before the formal proof, some definitions from [2,6] have to be reminded.

A finite failure-free input-first execution α is defined to be 0-valent if (1) some failure-free extension of α contains a $decide(0)_i$ action, for some process P_i, and (2) no failure-free extension of α contains a $decide(1)_i$ action. The definition of a 1-valent execution is symmetric. A finite failure-free input-first execution α is univalent if it is either 0-valent or 1-valent, and is bivalent if it's not univalent.

Lemma 3. $G(C)$ contains a hook.

Proof. Arbitrarily arrange all the tasks of C into an infinite sequence $\sigma = \sigma_1\sigma_2...\sigma_n...$ such that the process tasks all appear in prefix $\sigma_1\sigma_2...\sigma_n$. Let Σ_i=$\sigma_1\sigma_2...\sigma_{n+i}$, $i\geq 1$.

Now starting with a bivalent failure-free α_b), we construct a path π in $G(C)$ round after round.

In round i, for each $i \geq 1$, we consider the tasks in the segment Σ_i from left to right. Suppose that we have reached a bivalent execution α so far, and that task e is the next one in Σ_i that is applicable to α.

[Lemma 2, 2] implies that, for any finite failure-free extension $\alpha' = \alpha \cdot \gamma$ where e is not executed along the suffix γ, e is applicable to α', and hence $e(\alpha')$ is defined. We look for a vertex α' of $G(C)$, reachable from α without following any edge labeled with e, such that $e(\alpha')$ is bivalent. If no such vertex α' exists, the path construction terminates. Otherwise, we proceed to $e(\alpha')$ and then there are two possibilities. (1) If there a task e' after e in Σ_i which is applicable to $e(\alpha')$, consider e' in the next step. (2) If either e is the end of Σ_i or no task after e in Σ_i is applicable to $e(\alpha')$, then go to round $i+1$ and consider the left-most task e' in Σ_{i+1} that is applicable to $e(\alpha')$. In the second case, such e' in Σ_{i+1} always exists since Σ_{i+1} contains all process tasks which are always enabled by assumption.

For the detail of this construction, please refer to [Appendix I, 12].

We claim that π must be finite. Suppose for contradiction that it's infinite. Then infinitely many rounds occur in the construction. Given a task σ_i, it gets a turn to be executed in every round j such that $j \geq \max\{1, i\text{-}n\}$, so it gets infinitely many turns to be executed in π. As a result, π is a failure-free input-first fair execution of C. By the termination condition of consensus object, every process decides in π, which contradicts the facts that every finite prefix of π is bivalent.

The rest of the proof tries to find a hook structure following the finite execution . Since the counterpart of the proof of [Lemma 5, 2] doesn't care whether the collection of tasks is finite or infinite, it carries over. □

4 Impossibility for Systems with Shared Variables

In this section we further generalize Theorem 2 by replacing the canonical reliable registers with shared reliable variables. We show that Theorem 2 still holds if failure-aware services are not allowed. Based on the system model in Subsection 2.2, the following Theorem 3 is obtained.

4.1 Impossibility and the Idea for Its Proof

Theorem 4. $(f+1)$-resilient consensus objects can't be implemented from infinitely many shared reliable *variables* and canonical failure-oblivious f-resilient services.

Suppose for contradiction that there exists such an implementation. Lemmas 2-7, plus Claims 1-5 in the proof of [Lemma 8, 2], except Claim 3, all carry over. However, the proof of Claim 3 relies on the key fact that an invocation to a canonical register from a process P_i only affects the local state of the process and the ith invocation buffer of the register, leaving the register's value unchanged. On the contrary, an access to a shared variable instantaneously modifies its value. As a result, the proof of that Claim 3 can't carry over.

To circumvent proving that claim, we simulate systems with shared variables and canonical services by those with reliable registers and canonical services. Then apply Theorem 2, rather than follow its proof.

4.2 The Construction of a Simulating System

In fact, the simulation presented here is quite generic, because of the following characteristics. Firstly, the services are not necessarily canonical ones. Secondly, there is no constraint on the resilience of the services. Thirdly, the reliable atomic objects are required to be canonical. Fourthly and lastly, unlike [Theorem 13.7, 6], the existence of turn functions are not required.

However, some constraints on services are still needed. Arbitrarily chose a segment $\gamma = \alpha \cdot \beta$ from a (fair) trace of a service S. (1) If α is a response and β is an invocation, then replacing γ with $\beta \cdot \alpha$ still results in a (fair) trace of S. (2) If α and β are responses at different endpoints, then replacing γ with $\beta \cdot \alpha$ still results in a (fair) trace of S. (3) If $\alpha{=}fail_i$ for some endpoint i of S, and β is an arbitrary external action of S (not an invocation from endpoint i), then replacing γ with $\beta \cdot \alpha$ still results in a (fair) trace of S. (4) If $\alpha{=}fail_i$, there is an (fair) execution e of S such that τ=trace(e) and either α is the first action or immediately follows an external action in e.

These constraints aren't so restrictive, for example, any canonical failure-oblivious service satisfies all of them.

Now consider a shared variable system C with services, as specified in Subsection 2.2. The services are supposed to satisfy the above constraints. There are two technical assumptions on the processes. (1) In every state, each process has an action enabled. (2) There is a single task for each process, containing all non-input actions of the process. The two assumptions don't reduce the generality of our simulation (as in the sense of Theorem 5), since there must be such a system that simulates C.

Let V be the set of shared variables of C, and associate each v in V with a compatible reliable atomic object o_v. Atomic object o and variable v are said to be compatible if both of the following conditions hold. (1) o and v are of the same sequential type Γ_v. (2) A process P of C can directly access v if and only if P is in the endpoint set of o.

Let $O{=}\{o_v|v{\in}V\}$. We construct a system $T(C)$ which intuitively, is derived from C by replacing each $v \in V$ with o_v. The aim of $T(C)$ is to simulate C in some sense. The construction of $T(C)$ is specified as follows.

A shared variable system with services can be described as $\langle \mathfrak{P}, \mathfrak{I}, V \rangle$, where $\mathfrak{P}, \mathfrak{I}, and V$ are its set of processes, services, and shared variables, respectively. If $C{=}\ \langle \{P_1, P_2, ...P_n\}, \mathfrak{I}, V \rangle$, then $T(C)\ {=}\langle \{Q_1, Q_2, ...Q_n\}, \mathfrak{I} \cup O \rangle$. The processes Q_i is almost the same as P_i, but has the following difference in states, signature, and transition relation.

Q_i includes all the state components of P_i, plus seven more. (1) A binary **semaphore**, whether it's 1 indicates whether process Q_i is waiting for the response from an atomic object in O. (2) An 1-length queue **pending-invo**, storing a pending invocation to an object in O. (3) An 1-length queue **vari-resp**, storing the response just received from an object in O. (4) A variable **local-tran**, recording how to transit the local state of Q_i once the response from an object in O is received. (5) An infinite first-in-first-out queue **resp-buffer**, holding the responses sent by services in $\mathfrak{I}$ but not yet *processed* by Q_i. (6) A binary **flag**,

whether it's 1 implies whether it's Q_i's turn to process a response in the buffer. (7) A Boolean **failed**, whether it's True implies whether $fail_i$ has occurred. Initially, semaphore=0, pending-invo is empty, local-tran is arbitrary, vari-resp is empty, resp-buffer is empty, and flag=1. The introduction of the flag is a little technical, in order to preclude that Q_i keeps busy with only receiving and processing service responses in a fair execution.

Let Φ_i denote the set of internal actions of P_i that access a shared variables. Given an arbitrary $c \in \Phi_i$, introduce an action, denoted by $l(c)$, which is to start up c. Introduce another action $stub_i$ to perform the local part of all c. For each input action b of P_i, introduce an action $l(b)$, which is intended to *process* b, i.e. to change the local state of Q_i as b does that of P_i; the original b is preserved, while it means that P_i only receives the input. Introduce an internal action $live_i$, to keep Q_i live even it fails just after some $l(c)$. Let the input signature, output signature, and internal signature of P_i be *In*, *Out*, and *Int* respectively, and those of Q_i be In', Out', and Int' respectively. Then $In' = In \cup \{b|$ b is an output action at endpoint i of an atomic object in $O\}$, $Out' = Out \cup a | a \neq fail_i \wedge$ a is an input action at endpoint i of an atomic object in $O\}$, and $Int' =$(Int-Φ_i)$\cup \{l(c)| c \in \Phi_i\} \cup \{l(b)|b \in In \wedge\ b \neq fail_i\} \cup \{stub_i\}$.

To give an intuition on the transition relation of Q_i, we sketch here the idea of using $T(C)$ to simulate C.

Firstly, all the output actions and internal actions except those in Φ_i are simulated directly, with only their preconditions changed to involve the new state components.

Secondly, an action $c \in \Phi_i$ that accesses a shared variable is simulated by four steps of Q_i. The first step $l(c)$ starts up the simulation of c, enqueuing *pending-invo* with the invocation to v involved in c, and storing into *local-tran* the relation about how to change local state as in c. The second step invokes o_v as is hinted by *pending-invo*, and then dequeues *pending-invo*. The third step enqueues *vari-resp* with the response from o_v. The forth step $stub_i$ changes the local state according to *vari-resp* and *local-tran*, and dequeues *vari-resp*. The four steps are collectively called a complete simulation of c, $l(c)$ is celled the initialization, and $stub_i$ the finalization.

The four steps must be executed one after another without interruption, so other actions, even the inputs from services and external world, must be temporarily disabled. As a result, on the one hand, $l(c)$ sets semaphore to 1, $stub_i$ resets it to 0, the other actions keep it unchanged and most of them are enabled only if it's 0. On the other hand, each input action b$\neq fail_i$ of P_i is simulated in two steps, one (also named b in Q_i) to buffer the response, and the other (i.e. $l(b)$) to update the local state using the response. b and $l(b)$ don't have to be executed consecutively, and in fact, $l(b)$ is always done between some $stub_i$ and the next $l(c)$.

There are two technical maneuvers. On the one hand, in order for the simulation to preserve fairness of traces, it must be precluded that Q_i indefinitely performs only b and $l(b)$ for inputs b and ignores real workload. So, each $l(b)$ sets flag to 0 and is enabled only if flag=1, and flag can be reset to 1 only by

other input actions. On the other hand, to keep Q_i live, when it fails during the simulation of a shared variable action, semaphore must be reset to 0.

Please refer to [Appendix II, 12] for the formal definition of $T(C)$.

4.3 A Property of $T(C)$

If a system S is obtained by composing a collection of component automata and then hiding a set of actions, denote by $R(S)$ the system where the set of actions are not hidden.

Lemma 5. For $R(T(C))$ and $R(C)$, Theorems 8.2, 8.3, 8.5, and 8.6 in [6] hold, though each $fail_i$ action may be shared by infinitely many components. □

This guarantees that under some conditions, the (fair) traces/executions of the component automata can be pasted into (fair) traces/executions of the complete system.

Lemma 6. $T(C)$ simulates C in the following sense.

(1) They have the same interface,

(2) Any (fair) trace of $T(C)$ is a also a (fair) trace of C, up to a permutation which preserves both the order of invocations and that of responses at each port, and preserves the input-covering property. A trace is input-covering if at each port, an invocation precedes all responses.

Remark of the proof: The basic idea is similar to the proof of [Theorem 13.7, 6], but there are two key differences. Firstly, the existence of services, plus the non-existence of turn functions, makes it possible that a process Q_i) receives inputs (including responses) during its simulation of a variable access of P_i. Secondly, inputs sent to Q_i) are buffered and not necessarily processed immediately, so it's possible that Q_i) receives inputs or performs locally controlled actions during its simulation of an input to P_i. As a result, that proof can't carry over. Our proof of Lemma 6 is elaborated in [Appendix III, 12]. □

Suppose the above C is an implementation of n-process consensus object from shared variables and canonical f-resilient failure-oblivious services. A trace of C is said to be input-first if each P_i begins with an init() action, and has no other init() actions.

Corollary 7. There is a fair input-first trace of C where no more than $f+1$ processes fail, and which doesn't satisfy the three conditions of consensus.

Proof: Assume for contradiction that this is not the case. In the construction of $T(C)$, if each o_v is a canonical reliable register, then $T(C)$ is a system with canonical reliable registers and f-resilient failure-oblivious services. [2] in fact proves that there is a fair input-first trace α of C where no more than $f+1$ processes fail, and which doesn't satisfy the three conditions of consensus. By Theorem 5, there is a fair trace β of C which is the same as α up to a certain permutation. The property of the permutation guarantees that is also input-first, and that the projection of β to each port of C is the same as that of α

to each port of $T(C)$. As a result, no more than $f+1$ processes fail in β but β doesn't satisfy the three conditions of consensus. A contradiction. □

Corollary 6 immediately leads to Theorem 4.

5 Conclusion

We have generalized the results of [2] in two dimensions. First, we show that all the impossibility results still hold even if infinitely many reliable registers and fault-prone services are allowed. Second, we show that even if the system is strengthened by replacing canonical reliable registers with shared variables, the impossibility still holds in the case where canonical failure-oblivious services are used. To prove the second result, we in some degree generalize [Theorem 13.7, 6] by discarding the requirement of turn functions, and by extending shared variable systems to systems having both shared variables and failure-oblivious services. Our work under way is to extend our second result to the case of failure-aware services.

Acknowledgment

The work is supported by China's Natural Science Foundation (60603004, 60403023).

References

1. S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2):51-59, June 2002.
2. P. Attie, et al. The Impossibility of Boosting Distributed Service Resilience (Extended abstract). In Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS2005), 39-48. The full version is available at http://theory.lcs.mit.edu/tds/papers/Attie/boosting-tr.ps
3. M. Fischer, et al. Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(3):374-382, April 1985.
4. M. Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems, 13(1):124-149, January 1991.
5. P. Jayanti. On the robustness of Herlihy's hierarchy. In Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC1993), 145-157.
6. N. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, 1996.
7. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. Journal of the ACM, 46(6):858-923, November 1999.
8. M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM Journal on Computing, 29:1449-1483, March 2000.
9. M. Herlihy and S. Rajsbaum. Set Consensus Using Arbitrary Objects. In Proceedings of 13th Annual ACM Symposium on Principles of Distributed Computing (PODC 1994), pp. 324-333.

10. Y. Afek, et al. Computing with Faulty Shared Memory. In Proceedings of 13th Annual ACM Symposium on Principles of Distributed Computing (PODC 1992), pp. 47-58.
11. P. Jayanti, et al. Fault-Tolerant Wait-Free Shared Objects. Journal of the ACM, 45(3): 451-500, May 1998.
12. X. Liu, et al. Revisiting the Impossibility for Boosting Service Resilience. Technical report, January 1998. Available at http://blog.software.ict.ac.cn/xliu/files/2007/03/RevisitingtheImpossibilityfor BoostingServiceResilience-full.pdf