SIAM J. COMPUT.
Vol. 29, No. 2, pp. 351–377
© 1999 Society for Industrial and Applied Mathematics

SOLVABILITY IN ASYNCHRONOUS ENVIRONMENTS II: FINITE INTERACTIVE TASKS*

BENNY CHOR† AND LEE-BATH NELSON‡

Abstract. Identifying what problems can be solved in a given distributed system is a central question in distributed computing. In this series of works, we study this question in the context of asynchronous fault tolerant systems that can execute consensus. These systems can be those executing deterministic protocols with access to a consensus routine or those running randomized error-free protocols. A previous work handled the class of distributed decision tasks. In these tasks, each processor receives one local input and has to respond with one local output.

In an *interactive distributed task* each of n processors receives a sequence of local inputs and has to respond *on line* with an output for every new input (before getting its next input). Different processors can be at different stages concurrently, so that additional inputs are received by fast processors while slow processors are still working on early inputs. An interactive task is called finite if the number of local inputs (and outputs) is finite. Interactive tasks can neither be described as a single huge decision task nor be decomposed into distinct, independent decision tasks.

The main result of this work is an exact characterization of the finite interactive tasks which can be solved by t-resilient protocols in either of the above two models. The major tool we use in the characterization is a directed acyclic graph that is associated with an interactive task. Properties of this graph are used to determine the resiliency of the task and to devise a "generic" resilient algorithm which solves such tasks. This generic algorithm can be viewed as a repeated, deterministic reduction to a consensus subroutine. This implies that any finite interactive task is solvable by randomized error-free protocols iff it is solvable by deterministic protocols with access to consensus.

Key words. solvability, asynchronous distributed systems, fault tolerance, randomized algorithms, consensus, interactive tasks, decision tasks, adversary scheduler

AMS subject classifications. 68Q22, 90D06, 90D43

PII. S0097539795294979

1. Introduction.

1.1. Background. A central question in distributed computing is identifying what problems can be solved by a given distributed system. In typical systems, each one of n processors starts with some local input and communicates with other processors in order to produce globally meaningful outputs. If the system is perfect, in the sense that all processors are reliable and communication is error-free and is instantaneously relayed, then every well-defined task can be solved (assuming, as usual, that the processors are not computationally limited). However, perfect systems are either rare or nonexistent. Communication links tend to introduce errors and delays. Processors may become slow, stop operating, or even exhibit malevolent behavior.

One of the more popular models of distributed computing is the asynchronous crash failure model. Here, processors may crash without supplying any warning be-

*Received by the editors November 20, 1995; accepted for publication (in revised form) June 26, 1998; published electronically October 5, 1999. This research was supported by US–Israel BSF grant 88-00282. A preliminary version of this paper [14] appeared in *Proceedings of the* 10*th ACM Symposium on Principles of Distributed Computing*, 1991, pp. 37–49.

http://www.siam.org/journals/sicomp/29-2/29497.html

†Institute of Fundamental Sciences, Private Bag 11-222, Massey University, Palmerston North, New Zealand (B.Chor@massey.ac.nz).

‡Graduate School of Business, Stanford University, Palo Alto, CA 94305 (leen@leland.stanford.edu). The work of this author was done while at the Department of Computer Science, Technion, Haifa, Israel.

forehand. Asynchrony implies that there is no way to distinguish a failed processor from a very slow one. An alternative way to describe this is that there is an adversary scheduler that decides which processor moves when. Thus the scheduler controls the pace and failure of processors. A task is called *t-resilient* if it is solvable by a protocol, withstanding up to t processor crash failures. In addition to the important fault-tolerance aspect, crash resilient protocols have other merits: the more resilient the protocol, the less faster processors are delayed by waiting for slower ones. In particular, in a system with n processors, $n-1$ resilient protocols are *wait-free*—every processor can run at its maximum speed [19, 17].

A key result in this area was given by Fischer, Lynch, and Paterson [16], who have shown that the consensus problem is not even 1-resilient in the message passing model with respect to *deterministic* protocols. This result has been extended to the shared memory model [15, 21, 11]. (The conference version of the last reference proves this explicitly for a system of two processors.) This impossibility has motivated the study of *randomized* consensus protocols. A host of consensus protocols for various types of adversaries has been found for both the message passing model (e.g., [7, 24, 12, 6]) and the shared memory model (e.g., [11, 2, 1, 9, 3]). In particular, consensus has efficient *wait-free* (i.e., $n-1$ resilient) solutions in the shared memory model and $\lfloor \frac{n-1}{2} \rfloor$-resilient solutions in the message-passing model, even in the presence of a "strong adversary" [2, 6].

Consensus is an important problem, and in a certain sense it is a complete task, as follows from our results. Still, in order to understand the power and limitations of error-free randomization with respect to arbitrary distributed tasks, consensus alone does not suffice. To clarify this point, it is helpful to compare consensus with the *parity* task. In this latter task, each processor is required to output the XOR (parity) of all n inputs. Error-free randomization is powerful enough to overcome the coordination problems which prevent a deterministic solution to consensus, but it is of no help when facing problems of missing information. In the parity task, any irrevocable output by one processor (before knowing the inputs of all other $n-1$ processors) may later turn out to be inconsistent with additional inputs. Our work suggests a formal framework to capture these notions of missing and consistent/inconsistent inputs. Another aspect of fault tolerant models that is exemplified in this paper is that it is important for the adversary to have adequate powers. In particular, the adversary needs to be able to "resurrect" a processor, thereby turning a processor that previously may have appeared faulty into a merely slow one. Also, the adversary must be able to fail a processor that was previously active.

We study asynchronous fault tolerant systems that execute protocols of either of the following two types:

- deterministic protocols with access to consensus,
- randomized error-free protocols.

While we present most of our results in terms of *randomized error-free protocols*, the proof implies that in fact any finite interactive task is t-resilient in one model iff it is t-resilient in the other. In terms of what can and cannot be solved, it suffices to restrict the use of randomization to consensus. Thus if a consensus mechanism is built in, no randomization at all is needed. (Of course, randomization might still be used to speed up computations.)

This paper is the second in a series of three papers which study solvability by randomized error-free protocols, operating in asynchronous crash failure environments. The first paper in this series [13] deals with distributed decision tasks. The character-

ization there (as well as in the present work) is of combinatorial nature. Despite the added power of randomization, the characterizations and their proofs are fairly simple and yield effective procedures for testing solvability. This stands in sharp contrast to the recent works on deterministic solvability [8, 18, 27], which develop a methodology for characterizing decision tasks that are t-resilient with respect to deterministic protocols (*without* built-in consensus). These works are fairly complicated, use topological tools, and do not seem to yield effective characterization procedures. To the best of our knowledge, solvability of interactive tasks has not been addressed in the deterministic model.

A different line of research in deterministic solvability has dealt with *initial faults*. In the initial faults model, each processor either is initially crashed or remains active forever. Taubenfeld, Katz, and Moran [28] and Taubenfeld and Moran [29] have characterized t-resilience of distributed decision tasks with respect to deterministic protocols in the initial fault model. Interestingly, the characterization is the same as t-resilience with respect to randomized error-free protocols in the regular crash failure model [13]. This raises the question whether the two models have the same capabilities when richer classes of tasks are considered. In this work, we demonstrate a negative answer to this question. We describe a two-stage task that is solvable in the initial fault model but not in the regular crash failure model. The initial fault model enables the election of a "leader" who collects inputs and assigns outputs in a centralized fashion (and is not allowed to fail while doing this). While for the one round, decision tasks, randomization can be used to yield the effect of a leader, this is no longer the case when two-stage interactive tasks are involved. Intuitively, the models are different because in the initial faults model the adversary's powers are seriously limited.

1.2. Finite interactive tasks—motivation. Finite interactive tasks are an extension of distributed decision tasks [22, 13] (where each processor gets a single input and produces a single output). The number of rounds, namely the number of inputs received by each processor, is specified as part of the task. While one round corresponds to decision tasks, even tasks with two rounds constitute a nontrivial generalization. What distinguishes interactive tasks is the on-line character of the interaction. Each new local input is given only after the corresponding processor has irrevocably produced its previous output. Fast processors can work on late inputs while slower ones are still working on earlier inputs. For example, P_1 can work on its third input, P_2 on its fifth input, and P_3 on its first input concurrently. This implies that interactive tasks can neither be described as a single huge decision task nor be decomposed into distinct, independent decision tasks. Sequential systems as defined by Herlihy [17] and by Plotkin [23], such as stacks and queues, can also be formulated as interactive distributed tasks. (Finite interactive tasks will model such systems that are restricted to a bounded number of accesses per processor.)

We give several examples that should help clarify the expressive power of finite interactive tasks. Consider a multiround task where in each round the processors have to select a unique leader among all candidates. We indicate a processor's candidacy, in a given round, by inputting 1, while 0 indicates that the processor is not a candidate. The selected processor outputs 1 in the appropriate round, while all others output 0. (If there are no candidates in a given round, then nobody is selected.) So far, this leader selection problem can be viewed and solved as a collection of independent decision tasks. However, it may be useful to prevent a fast processor from taking permanent control of the system. Thus we add the requirement that no processor

is selected twice. This gives rise to different finite interactive tasks, characterized by the total number of "selection campaigns" (or, equivalently, rounds). In the kth task in this list each processor has k inputs and k outputs. In this revised task, a processor that was selected in some round will output 0 in all subsequent rounds, *even if there are no other candidates*. It is not hard to see (and follows easily from our characterization) that the k round task is $(n-1)$-resilient for any $k \geq 1$. (The task is well defined for any k, although after everyone is selected, it becomes quite boring.)

The second example is a variant of the first. Here, in every round every processor receives as input a "priority," which is a positive integer in case the processor is a candidate and 0 if it is not. A leader can be selected if there are at least ℓ other processors with smaller priorities, among all processors that were not chosen already. (If nobody is interested, then every relevant processor gets a 0 and all inputs are among the least ℓ priorities, and in this case everyone outputs a 0 to indicate nonleadership.) As before, the unique selection is indicated by a 1 in the appropriate output. The selection process is repeated k times. How resilient is this task? For $k = 1$, a prospective leader must see at least ℓ other inputs before "fighting" for candidacy. This argument can be formalized to see that for $k = 1$, the task is $(n - \ell - 1)$-resilient. Now consider $k = 2$. At the second round, the prospective candidate must still wait for ℓ other processors, but now these could come from a pool of only $n - 1$ processors. This argument can be formalized to see that the task now is $(n - \ell - 2)$-resilient. For general k and ℓ, it can be shown that this finite interactive task is $(n-\ell-k)$-resilient. This example can be extended further to describe cases where the selection of the ith round leader may depend on inputs from the $(i + 1)$st round (e.g., if future inputs of previous leaders can act as tie breakers for the choice of the current leader), and so on.

As a final example, we now describe a different task, which can be viewed as a resource allocation problem. There are m resources ($m < n$), denoted by 1 through m. At each of k rounds, any number of no more than m of the n processors can announce their interest in getting one of the resources (it does not matter which specific resource) by receiving an input of 1 (otherwise the processor gets 0). If the processor is allocated resource number j, then its output for that round is j. If no resource is allocated, the processor's output is 0. Once a resource is allocated, the processor can retain it by continuing to get input 1. To release a resource, the processor must get input 0. The task specifies that no resource is allocated at the same round to more than one processor. In addition, all requests should be granted. This task is related to a continued renaming problem [5]. For $k = 1$, it is not hard to see that the task is $(n - 1)$-resilient (notice the difference from the deterministic case where the name range has to be larger than m). But this is no longer the case if $k > 1$. Consider a fast processor who was not interested in a resource at round 1 but became interested in one at round 2. This processor cannot simply grab a resource, despite the fact that it is guaranteed that such resource will eventually be freed for round 2: up to m tardy processors can wake up and ask for resources at the first round, and only one of them may release its resource in the second round. Our fast processor must wait until at least $m - 1$ of the first round users announce their input, and only then can the processor grab a resource. This will be either one of the resources released by one of the $m - 1$ processors or the remaining resource, in case all these processors retained their resource. This implies that if m resources were claimed at round 1, the processor may have to wait for the second round inputs of $m-1$ of them before giving an output in the second round. This argument leads to realizing that for $k \geq 2$, this

resource allocation problem is 1-resilient but not 2-resilient.

The examples we give demonstrate the versatility of the finite interactive task formulation. We conclude that this is a rich class of tasks which constitutes a meaningful extension to the class of distributed decision tasks.

1.3. Highlights of the characterization. Denote by ISM_t (resp., IMP_t) the class of finite distributed interactive tasks that are solvable in the asynchronous shared memory (resp., message passing) model by a terminating t-resilient randomized protocol, which never errs and works in the presence of a strong adversary scheduler [2, 6]. (A protocol is terminating if each processor stops participating and halts after producing its last output.) Our main results are necessary and sufficient combinatorial conditions which determine membership in ISM_t (for $0 \leq t < n$) and IMP_t (for $0 \leq t \leq \lfloor \frac{n-1}{2} \rfloor$). A similar characterization, for nonterminating protocols, is also given. These results subsume the previous results of [13], which characterized resiliency of distributed decision tasks. We remark that the proof methods in the present work are substantially more involved than those in [13], due to the more general nature of interactive tasks.

We show that what determines resiliency in the randomized error-free model is the amount of information available (at every possible step) to the active processors and whether this information suffices to make moves that are compatible with every potential future development. In order to capture these properties, we associate every finite interactive task T with a directed acyclic graph (DAG), whose nodes represent states of the distributed system. It turns out that the DAG is a convenient tool with which to express the properties we need. In this subsection we outline how the DAG is defined and used for the characterization (while omitting some of the noncrucial details).

The nodes in the DAG contain partial vectors of (multi) input and (multi) output values that are globally known in the system. (For a concrete example, see section 4.) The nodes are first examined according to their "legality" and "consistency." Legality depends only on the indices in the partial vectors (these are the "S vectors" of section 3) and not on the values themselves. It means that no processor has produced an output in a given round before producing all the outputs of earlier rounds and getting all the inputs of earlier and the current rounds. Consistency is related to the task T and does depend on the values in the partial vector (these are the "Q vectors" of section 3). It means that for every possible completion of input values there is a corresponding completion of output values, such that the complete multi-input multi-output vector belongs to the task T.

We put a directed edge from node v_1 to node v_2 in the graph if v_2 extends the values in v_1 and contains exactly one additional input value. (It may contain one additional output, several additional outputs, or none.) We partition the nodes in the DAG into equivalence classes. Two nodes are called *equivalent* if they have the same sets of indices of revealed *inputs* (output indices do not matter here). This definition is useful in situations when there are directed edges from v to both u_1 and u_2 and the latter two nodes are *nonequivalent*. This implies that u_1 and u_2 extend v in different input indices, implying that two different processors have read an additional input in the corresponding moves.

We further refine each equivalence class and say that two equivalent nodes are *input equal* if the *values* of their revealed inputs are the same (output indices and values may still differ). With these definitions, we can describe the notion of a *t-founded* node (relative to the task T). This notion is central to our characterization.

Essentially, we think of a t-founded node as a "good" node from which there exists a strategy to advance while facing up to t crashes without getting stuck or making errors (relative to the task T).

The definition of t-founded is recursive, and we will now briefly describe its main points while omitting some special cases that correspond to the "boundaries" (where at least $n-t$ processors have already terminated). A node v in the DAG is t-founded if it is either a complete multi-input multioutput vector that belongs to the task T or a nonboundary partial vector which satisfies the following condition: there are at least $t+1$ nonequivalent nodes in the DAG, $u_1, \ldots, u_t, u_{t+1}$, such that there is a directed edge from v to each u_j, and all u_j's are t-founded. In addition, for every u with a directed edge from v to u there exists an input equal node u' that also has a directed edge from v to it, and u' is t-founded. (The definition of t-founded for boundary nodes is slightly modified, especially in the requirement for the number of nonequivalent sons.)

Having $t+1$ nonequivalent sons of v essentially implies that even if an adversary scheduler crashes t processors, the remaining ones can still make progress. The second condition (regarding input equal sons) guarantees that the system can advance along a path of consistent nodes (with respect to the task T) no matter which processor is active next. The main theorem states that a finite interactive task T has a t-resilient protocol iff the root of the DAG (the partial vector with no inputs and no outputs) is t-founded.[1]

We show that the condition is necessary by the following argument. Suppose T is a task where the root of the DAG is not t-founded and $\mathcal{A}$ is an algorithm for the task that is claimed to withstand t crashes. We demonstrate a strategy for an adversary scheduler that enables it to force the system to advance (with positive probability) along a path of nodes that are not t-founded. We show that such a path leads to an inconsistent node. This means that the adversary can force the algorithm $\mathcal{A}$ to err (with nonzero probability). Notice that in order to be able to force the processors down this path, the adversary needs to be able to fail a previously active processor as well as cause the processors to think that a slow processor may be faulty.

To show that the condition is sufficient, we design a generic protocol which guarantees that all processors take a coordinated walk along a path of t-founded nodes in the DAG. When all processors terminate, this path must lead to a complete vector that is in T. Such coordination is achieved by applying a consensus subroutine to every step in the walk.

We use randomized consensus algorithms as given by [2, 1, 3, 9, 26] for the shared memory model and by [6] for the message passing model. It is interesting to observe that randomization is needed only for consensus and gives no extra power beyond that. This follows from our generic algorithm, which can be viewed as a repeated *deterministic reduction* of an arbitrary interactive task to consensus (which itself requires randomized solutions). Therefore, our characterizations also can be applied to deterministic systems that have a built-in consensus mechanism, and in this case no randomization at all will be needed (see section 7 for additional discussion).

1.4. Organization. The remainder of this paper is organized as follows. Section 2 describes the computation models. In section 3 we formally define interactive tasks and related notions used in this paper. In section 4 we describe the DAG associated

[1] As stated, the theorem holds for the shared memory model. A slight variation, which corresponds mainly to the final stages of an execution, makes it applicable to the message passing model (the variation is termed *t-valid*).

with an interactive task. This graph plays a central role in section 5, where the characterization theorem for the shared memory model is stated and proven. Section 6 contains the characterization for the message passing model. Finally, in section 7 we present some concluding remarks, including the equivalence of error-free randomized model and the deterministic model with access to consensus.

2. The models. In this section we define the models of asynchronous computation which we use in the sequel, as well as the class of appropriate schedulers (or adversaries) that control our systems.

An asynchronous concurrent system is a collection of n processors. Each processor, P, is a (not necessarily finite) state automaton with an internal input register in_P and an internal output register out_P. The set of all states of the processor P is denoted by S_P. The input register contains a value v taken from a set IN, while the output register has initially the value $\perp$. The value in the output register can be changed to any value in the set OUT ($\perp \notin OUT$). For every input value that the processor gets, it must change the value of the output register (possibly to the same value as before) exactly once before receiving the next input or terminating (after the last input).

The two models we consider differ in the way that processors communicate among themselves. In the *shared memory model*, processors communicate via *shared registers*. Every shared register r is associated with a set of processors R_r, $\mid R_r \mid > 0$, that can read from the register and a set of processors W_r, $\mid W_r \mid > 0$, that can write into the register. These registers are atomic with respect to the read and write operations. (Although our protocols use only the simpler, multireader single-writer registers, the necessity of our conditions holds in the more general setting, presented here.)

In the second model, the *message passing model*, there is a communication link between every two processors. Processors may send and receive messages via these links. The links are atomic with respect to the send-message and receive-message operations, but there is no guarantee on the order in which the messages are received once they are sent.

Processors execute their programs by taking steps. An atomic step consists of one of the following:

1. An internal operation, possibly involving coin tosses.
2. Getting information from other processors (reading from a shared register in the shared memory model, or receiving a message from a link in the message passing model).
3. Giving information to other processors (writing into a shared register in the shared-memory model, or sending via a link in the message passing model).
4. Getting a new local input.
5. Giving a new local output.

Formally, every processor P takes steps according to its *transition function*, T_P. In case the step taken was a read (or receive), the new state of P depends not only on its old state but also on the value read (received) by this action. The transition function T_P could be either deterministic or nondeterministic. In the latter case, the actual step taken is decided via coin tosses (in a nondeterministic step the only difference between the old and new states is the coin toss, and no communication occurs in such a step). Given an asynchronous system as specified above, a *protocol* is a collection of n transition functions $T_1, \ldots, T_n$, one per processor.

A *configuration* C of the system, in the shared memory (message passing) model, consists of the state of each processor together with the contents of the shared registers

(communication links). In an *initial configuration*, every processor is in an initial state, and all shared registers and output registers contain the default value $\perp$ (all the links are empty). The set of all configurations will be denoted by $\mathcal{C}$. A *step* takes one configuration to another by activating a single processor P. A *run* of length ℓ is a sequence of ℓ steps. Each run has an associated *schedule* which is a sequence of ℓ processors, numbered according to the order of processors that take steps in that run. We denote schedules, finite or infinite, by a list of processor numbers, e.g., $(2, 3, 3, 2, 1)$. We say that processor P is activated k times in a run if P appears k times in the run's schedule. The *history* $\mathcal{H}$ of a run is the sequence obtained by interleaving the sequence of configurations with the steps in the run, starting with the initial configuration. For a finite run, we refer to the last configuration in its history as the *current* configuration.

A *scheduler* $\mathcal{S}$ is a mapping from $\mathcal{H}$ into the set of n processors. Given the configuration of the system, the scheduler picks the next processor that is to take a step. The scheduler could be either a deterministic mapping or a randomized one. The scheduler can be viewed as an adversary which tries to prevent the system from reaching its goal. Under this definition, the adversary is very strong: it has complete knowledge of the state of every processor and of the contents of the shared registers (communication links) during the entire history. In case the processors are randomized, the scheduler could also base its choices on the outcome of past coin flips as well as the current coin flip (if the next step is randomized). We do not allow it, though, to be able to predict *future* randomized moves of the processors. This is a necessary requirement if randomization is to be helpful at all, and it is used in all algorithms where randomization is employed, e.g., [25, 20, 7]. In addition, the scheduler picks the inputs to be given to the processors (from the possible legal inputs).

Finally, we note that both models are asynchronous, meaning that there is no global clock in the system and each processor runs at its own pace. In the message passing model this also means that messages can be delivered with arbitrary delays (and possibly out of order).

In each one of these models there are two submodels according to the nature of the processors. All the processors can be of one of two types:

1. Terminating processors that terminate and halt once their final output has been given.
2. Nonterminating processors that continue to operate even after giving their final output. These processors may help and coordinate among the slower processors that have not yet finished their task.

The second type of processor yields a more robust system, which could solve any task solvable by processors of the first type.

In this work we study the resiliency of tasks in these two models.[2] There is a resiliency parameter, t, that denotes the extent of resiliency required. The *t-resiliency* requirement imposes restrictions both on the scheduler and on the protocol. In every infinite schedule under the scheduler, S, at least $n - t$ processors will be activated infinitely many times. We will call such a scheduler a *t-bounded scheduler*. Intuitively, this means that the scheduler may fail-stop at most t processors.

A *round* of a t-bounded schedule is a minimal schedule of the processors (starting from any configuration) in which at least $n - t$ processors are scheduled at least once.

[2]Most of our results are developed for terminating processors. Once these results are obtained, the case of nonterminating processors is easy to handle. The results for the nonterminating model are stated in section 7.

Each processor which is scheduled in the round is said to be *active* in it. Notice that after $n - t$ processors have terminated, a schedule in which all the terminated processors are scheduled (and thus no move will be made) is also a round.

DEFINITION 1. *We say that an algorithm, A, is a t-resilient algorithm for a task, T, if the following conditions hold:*

- *Safety. For every legal input for T, for any t-bounded scheduler, S, and for any execution of A, if all n processors terminate (or in the nonterminating models, if all n processors have given their last output), then the resulting output is correct with respect to the input for T (see Definition 2).*
- *Liveness. There is a constant, M, such that for every processor the* expected *number of rounds in which it is active, before producing its final output, is bounded by M. For terminating processors this requirement implies that after $n-t$ processors have terminated, the expected number of* steps *taken by each of the remaining t processors in order to terminate is bounded by M (no matter how the steps of this individual processor are interleaved with steps of other processors).*

It should be noticed that we require that protocols never err. Although there could be a positive probability for very long nonterminating runs, this probability should be very small (converging to 0 with the length of the run).

3. Basic definitions. In this section we define finite interactive tasks and what it means for such a task to be t-resilient. We introduce the notation of partial vectors and use this notation to express states of systems that implement interactive tasks and transitions between such states.

Distributed interactive tasks are a generalization of distributed decision tasks. A decision task is a collection of input-output pairs (each pair is an n-vector). Analogously, we define an *interactive$_k$ task* as a collection of k pairs. Each element of every pair is an n-vector. The first element represents an input vector and the second an output vector. The following formulation turns out to be convenient for our purposes.

DEFINITION 2. *An interactive$_k$* task *(or k-*stage distributed interactive task*), T, is a collection of $2k$-tuples of the form $\langle I^1, \ldots, I^k, O^1, \ldots, O^k \rangle$. Each I^j and O^j is an n-component vector over an arbitrary alphabet, corresponding to the n inputs and n outputs of the jth stage, respectively. The* legal inputs *for T (denoted $IN(T)$) are all the k-tuples of n-component vectors $\langle I^1, \ldots, I^k \rangle$ such that there exists $\langle O^1, \ldots, O^k \rangle$ for which $\langle I^1, \ldots, I^k, O^1, \ldots, O^k \rangle \in T$.*

When an execution of an interactive$_k$ task starts, a processor P_i gets the input for the first stage. After it submits the output for this stage (which is irrevocable) it receives the input for the second stage, and so on. In general, each processor, P_j, gets its input for stage $i + 1$ only after submitting its output for stage i, for all $1 \leq i \leq k - 1$. Different processors may be in different stages simultaneously. At a certain point during an execution, it is possible that there will already be, for example, seven outputs for the first stage, five for the second, two for the third, and none for the fourth.

We will represent the states in an instance of an algorithm using partial vectors which contain input and output values. If we take a vector $\vec{V}$ of length n over a certain alpha-beth, Σ, and a set of indices, J, where $J \subseteq \{1, \ldots, n\}$, then the partial vector $\vec{V}_J$ is defined as follows:

- If $i \in J$, then $V_J(i) = V(i)$, the ith coordinate of $\vec{V}$.
- Otherwise $V_J(i) = \perp$, where $\perp$ is an agreed sign for "don't know yet."

This means that we know only the values of $\vec{V}$ at the indices indicated by J. Typically,

$\vec{V}_J$ is compatible with more than one vector $\vec{V}$, namely, there may be a $\vec{V}'$ such that $\vec{V}' \neq \vec{V}$ but still $\vec{V}'_J = \vec{V}_J$. For example, the partial vector $\vec{V}_\phi$ is compatible with every vector of the same length.

One property of a $2k$-tuple, representing the state of an interactive$_k$ task, corresponds to the temporal order in which input and output indices are revealed. This property is formalized in the next definition, which uses the notion of *corresponding partial input and output*. We say that a partial input and output $Q = \langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle$ corresponds to the $2k$-tuple $S = \langle A_1\ ,\ \ldots\ ,\ A_k\ ,\ B_1\ ,\ \ldots\ ,\ B_k\rangle$ if the revealed indices in Q match the index-sets in S.

DEFINITION 3. *Let $S = \langle A_1, \ldots, A_k, B_1, \ldots, B_k\rangle$ be a $2k$-tuple of index-sets. S will be called a* legal 2k-tuple *if the following conditions are satisfied:*

- *For all i, $1 \leq i \leq k$, it holds that $B_i \subseteq A_i$ and $A_i, B_i \subseteq \{1, ..., n\}$.*
- *For all i, $1 \leq i \leq k-1$, it holds that $A_{i+1} \subseteq B_i$.*

Let Q be a partial input and output vector, corresponding to S. We say that Q is a legal partial input–output vector *if S is a legal $2k$-tuple.*

The requirements ensure that S represents a possible state of revealed inputs and outputs in an instance of an algorithm for an interactive$_k$ task. By demanding that $B_i \subseteq A_i$ we guarantee that no processor will produce its ith output before receiving its ith input. By demanding that $A_{i+1} \subseteq B_i$ we guarantee that no processor will get its $(i+1)$st input before producing its ith output. However, the legality requirement does not suffice. We also want to guarantee that the "input output contents" matches a state that can fit the requirements of the task itself. This property is stated in the next definition.

DEFINITION 4. *Let $Q = \langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle$ be a legal $2k$-tuple of partial vectors. Q will be called* T-consistent *if for every extension I^1 to I^k, of $I^1_{A_1}$ to $I^k_{A_k}$ such that $\langle I^1, \ldots, I^k\rangle \in IN(T)$, there exist extensions O^1 to O^k, of $O^1_{B_1}$ to $O^k_{B_k}$ such that $\langle I^1\ ,\ \ldots\ ,\ I^k\ ,\ O^1\ ,\ \ldots\ ,\ O^k\rangle \in T$.*

We will say that output o^i_j is a consistent output *for the $2k$-tuple $\langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}$, $O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle$ if the $2k$-tuple $\langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B'_1}\ ,\ \ldots\ ,\ O^k_{B'_k}\rangle$, is T-consistent, $B'_k = B_k$ for every $k \neq i$, $B'_i = B_i \cup \{j\}$, and the jth coordinate of $O_{B'_i}$ equals o^i_j.*

An inconsistent $2k$-tuple corresponds to a state where the scheduler, by giving legal inputs, can force the algorithm to err. Thus, intuitively, consistency is a property that the processors wish to maintain.

Let us now formalize the notion of a "one-step" advancement from a legal $2k$-tuple of partial input-output vectors to another legal one. Intuitively, a one-step advancement corresponds to one additional input read by one processor and possibly a resulting extension of one or more outputs. If the new input is from the ith stage $(i \geq 2)$, then, by our convention, the reading processor has already given its $(i-1)$st output. There is an exception to this intuition, which occurs toward the end of a run, when only t or fewer processors still have unrevealed inputs. In this case, a single additional output is also considered an advancement, as the remaining t or fewer processors might be faulty and thus we cannot insist on input extensions here. We first formally define these early and late parts in a run.

DEFINITION 5. *The earlier parts of a run of an algorithm, where the set of processors that have already read their last input, A_k, satisfies $|A_k| < n - t$, are called the* main phase. *The final parts of a run of an algorithm, where $|A_k| \geq n - t$, are called the* concluding phase.

The definition of a one-step advancement is as follows.

DEFINITION 6. *Let*

$$S = \langle A_1, \ldots, A_k, B_1, \ldots, B_k \rangle, S' = \langle A'_1, \ldots, A'_k, B'_1, \ldots, B'_k \rangle$$

be two legal $2k$*-tuples and let*

$$Q = \langle I^1_{A_1}, \ldots, I^k_{A_k}, O^1_{B_1}, \ldots, O^k_{B_k} \rangle,$$

$$Q' = \langle I^1_{A'_1}, \ldots, I^k_{A'_k}, O^1_{B'_1}, \ldots, O^k_{B'_k} \rangle$$

be partial inputs and outputs corresponding to S *and* S'*, respectively. The pair* $[S', Q']$ *will be called a* t-subsequent *of the pair* $[S, Q]$ *if the following three conditions hold:*

- *For all* i, $1 \leq i \leq k$ *it holds that* $B_i \subseteq B'_i$.
- *Exactly one of the following is true:*
  1. *There is exactly one* i $(1 \leq i \leq k)$ *and one* j $(j \in \{1, \ldots, n\})$ *such that* $j \notin A_i$, $j \in B_{i-1}$, $A'_i = A_i \cup \{j\}$, *and for all* $m \neq i$ *it holds that* $A'_m = A_m$.
  2. *The system is in the concluding phase* $(|A_k| \geq n - t)$ *and there is exactly one* i $(1 \leq i \leq k)$ *and one* j $(j \in \{1, \ldots, n\})$ *such that* $j \notin B_i$, $B'_i = B_i \cup \{j\}$, *for all* $m \neq i$ *it holds that* $B'_m = B_m$, *and for all* $1 \leq \ell \leq k$ *it holds that* $A_\ell = A'_\ell$. *This corresponds to a single new output, given by* P_j *in the* i*th stage.*
- Q' *is an extension of* Q *and there exists* $< I^1, \ldots, I^k > \in IN(T)$ *such that* $< I^1, \ldots, I^k >$ *is an extension of* $< I^1_{A'_1}, \ldots, I^k_{A'_k} >$.

When $[S', Q']$ *is a* t*-subsequent of* $[S, Q]$ *and* j *is the unique index extending* A_i *(in case* 1*) or* B_i *(in case* 2*), we say that* P_j *is the* processor associated with the t-advancement.

The number of possible failures, t, is a parameter in the definition. An additional input is considered an advancement in both the main and the concluding phases, while an additional output (on its own) is considered an advancement only in the concluding phase. Notice that this definition can be used independently of consistency.

In [17], Herlihy studied concurrent implementations of sequential data objects. Finite versions of such objects may be represented as interactive$_k$ tasks. This can be done by taking the set of all possible sequences of correct operations on the object. For example, consider a concurrent implementation of a stack. The processors in this implementation are servers that receive in their input one of two possible requests:

- *push(value)*: push the parameter *value* onto the top of the stack and return (output) ℓ, where ℓ is the level of the stack in which the value was put (1 when the stack contains one element, 2 when the stack contains two elements, etc.). In our example the stack level is required as output mainly for didactic reasons.
- *pop*: pop the top of the stack and return (output) its value ("empty" if the stack is empty).

Obviously the outcome of a request given to a processor may depend on previous requests to some of the other processors. We will use the example of a three-processor stack as implemented by an interactive$_2$ task (two requests per processor) to illustrate the definitions. The resiliency parameter will be $t = 2$. Assume that in the first stage P_1 will be requested to push 5, P_2 will be requested to pop, and P_3 will be requested to push 7. These will be the inputs (but the processors do not know them initially). From the initial configuration, where no inputs and no outputs are known, we can

advance in one step to a state where one input is known (e.g., P_1 has "push 5" as input) or to a state where one input and one output are known (for instance, P_1 with the input "push 5" and with output "1," or P_2 with input "pop" and output "empty").

4. The corresponding DAG. In this section we define the DAG associated with an interactive$_k$ task. A run of an algorithm can be viewed as a sequence of $2k$-tuples of indices with their corresponding $2k$-tuples of partial inputs and outputs. We associate this sequence with a path in a DAG. There is a directed edge from v to u if u is a possible state representing a one-step advancement from v (namely, u is a t-subsequent of v). More formally, we have the following definition.

DEFINITION 7. *Let T be an interactive$_k$ task. The* DAG associated with T, *which we denote by $D(T)$, has the set of nodes*

$$V = \{[S,Q] | S = \langle A_1\ ,\ \ldots\ ,\ A_k\ ,\ B_1\ ,\ \ldots\ ,\ B_k\rangle, Q = \langle I^1_{A_1},\ \ldots,\ I^k_{A_k},\ O^1_{B_1},\ \ldots,\ O^k_{B_k}\rangle,$$

S is a legal $2k$-tuple, Q corresponds to S, and there exists $\langle I^1, \ldots, I^k\rangle \in IN(T)$ such that $\langle I^1, \ldots, I^k\rangle$ is an extension of $\langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\rangle\}$. The set of directed edges, E, is the set of all (v_1, v_2) where $v_1 = [S_1, Q_1]$ and $v_2 = [S_2, Q_2]$, such that v_2 is a t-subsequent of v_1 and v_2 is T-consistent.

The root of the DAG is the node $[\langle\phi, \ldots, \phi, \phi, \ldots, \phi\rangle, \langle I^1_\phi, \ldots, I^k_\phi, O^1_\phi, \ldots, O^k_\phi\rangle]$. The leaves (nodes from which there are no outgoing edges) are either nodes containing a complete $2k$-tuple of indices ($A_1 = \cdots = A_k = B_1 = \cdots B_k = \{1, \ldots, n\}$) with the corresponding full inputs and outputs, or nodes whose $2k$-tuple of indices is incomplete but do not have T-consistent sons. In general, we think of a node in the DAG as representing a possible state of the published input and agreed-upon output values at some point during a possible execution. An edge represents a transition according to the t-subsequency relation. Notice that one can "label" an edge from u to v according to the processor P_j which is associated with the t-advancement.

In our example of the stack, there will be an edge from the node representing the initial state, $[\langle\phi\ ,\ \phi\ ,\ \phi\ ,\ \phi\rangle, \langle I^1_\phi\ ,\ I^2_\phi\ ,\ O^1_\phi\ ,\ O^2_\phi\rangle]$, to the node representing the state where P_1 has seen the input "push 5," which is the node $[\langle\{1\}, \phi, \phi, \phi\rangle, \langle\{\text{"push(5)"}, \perp, \perp\}, I^2_\phi, O^1_\phi, O^2_\phi\rangle]$. There will also be an edge from the initial node to the node representing the state where P_2 has seen "pop" and returned "empty." This node is $[\langle\{2\}, \phi, \{2\}, \phi\rangle, \langle\{\perp, \text{"pop"}, \perp\}, I^2_\phi, \{\perp, \text{"empty"}, \perp\}, O^2_\phi\rangle]$.

The DAG represents all the possible advancements for all possible inputs. It is useful to group the vertices into equivalence classes according to the amount of information in them, disregarding the actual input and output values. This leads to the definition of *t-equivalent* nodes in the DAG.

DEFINITION 8. *Let*

$$v = [\langle A_1\ ,\ \ldots\ ,\ A_k\ ,\ B_1\ ,\ \ldots\ ,\ B_k\rangle, \langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle]$$

and

$$v' = [\langle A'_1\ ,\ \ldots\ ,\ A'_k\ ,\ B'_1\ ,\ \ldots\ ,\ B'_k\rangle, \langle I^1_{A'_1}\ ,\ \ldots\ ,\ I^k_{A'_k}\ ,\ O^1_{B'_1}\ ,\ \ldots\ ,\ O^k_{B'_k}\rangle]$$

be two nodes in the DAG corresponding to an interactive$_k$ task, T. These two nodes will be called t-equivalent *if the following two conditions are satisfied:*

- *For all i, $1 \le i \le k$, it holds that $A_i = A'_i$.*
- *If $|A_k| \ge n - t$, then for all i, $1 \le i \le k$, it holds that $B_i = B'_i$.*

In the equivalence relation we are concerned only with the *indices* of the revealed inputs/outputs, while the values in these indices do not matter. From the definition it follows that two nodes are *non-t-equivalent* in one of two cases: either they do not have the same indices of input revealed or they are both in the concluding phase, and their revealed output indices are different (for some i it holds that $B_i \neq B'_i$). As with the definition of t-subsequents, we take the outputs into consideration only in the concluding phase.

In our example, when examining whether the task is 2-resilient ($t = 2$), the nodes

$$v_1 = [\langle\{1,2,3\},\phi,\{1,2,3\},\phi\rangle,$$
$$\langle\{\text{“push(17)”},\ \text{“push(27)”},\ \text{“push(37)”}\}, I^2_\phi, \{1,3,2\}, O^2_\phi\rangle]$$

and

$$v_2 = [\langle\{1,2,3\},\phi,\{1,3\},\phi\rangle,$$
$$\langle\{\text{“push(57)”},\ \text{“push(67)”},\ \text{“push(87)”}\}, I^2_\phi, \{2,\perp,1\}, O^2_\phi\rangle]$$

are 2-equivalent (both nodes have $A_1 = \{1,2,3\}, A_2 = \phi$). While the nodes

$$u_1 = [\langle\{1,2\},\{2\},\{1,2\},\phi\rangle,$$
$$\langle\{\text{“push(17)”},\ \text{“push(27)”},\perp\},\{\perp,\text{“pop”},\perp\},\{2,1,\perp\}, O^2_\phi\rangle]$$

and

$$u_2 = [\langle\{1,2\},\{2\},\{1,2\},\{2\}\rangle,$$
$$\langle\{\text{“push(17)”},\ \text{“push(27)”},\perp\},\{\perp,\text{“pop”},\perp\},\{1,2,\perp\},\{\perp,27,\perp\}\rangle]$$

are *non*-2-equivalent. (These nodes have $|A_2| = 1 = 3 - 2$, but their B_2 sets are not the same.)

Notice that two nodes (collections of partial vectors) with the same input indices are revealed, but different values revealed in these indices *are* equivalent. The reason they are defined as equivalent is that in a specific run we are interested only in the input for this run, which cannot be compatible with both. We refine the t-equivalence relation, thereby partitioning the equivalence classes into subclasses, such that all the nodes in the same subclass will have the same *input values* (not just indices). The nodes in this subclass will be called *input-equal*. Input-equal nodes may differ both in their output indices and in their output values (provided they remain t-equivalent).

DEFINITION 9. *Let*

$$v = [\langle A_1,\ \ldots,\ A_k,\ B_1,\ \ldots,\ B_k\rangle, \langle I^1_{A_1},\ \ldots,\ I^k_{A_k},\ O^1_{B_1},\ \ldots,\ O^k_{B_k}\rangle]$$

and

$$v' = [\langle A'_1\ ,\ \ldots\ ,\ A'_k\ ,\ B'_1\ ,\ \ldots\ ,\ B'_k\rangle, \langle I^1_{A'_1}\ ,\ \ldots\ ,\ I^k_{A'_k}\ ,\ O^1_{B'_1}\ ,\ \ldots\ ,\ O^k_{B'_k}\rangle]$$

be two nodes of the DAG corresponding to an interactive$_k$ task, T. Let t be a resiliency parameter. The nodes v and v' will be called input-equal if the following conditions hold:

- *v and v' are t-equivalent.*
- *For all i, $1 \leq i \leq k$, it holds that $I^i_{A_i} = I^i_{A'_i}$ (which implies $A_i = A'_i$ as well).*

In the example of the stack, assume that the inputs in the first stage are "push 21" for P_1, "push 22" for P_2, and "push 23" for P_3. Now suppose that the input in the second stage for all three processors is "pop." In this situation, the nodes

$$v_1 = [\langle \{1,2,3\}, \{1,2,3\}, \{1,2,3\}, \{2\} \rangle, \langle \{\text{"push(21)"}, \text{"push(22)"}, \text{"push(23)"}\}, \{\text{"pop"}, \text{"pop"}, \text{"pop"}\}, \{2,1,3\}, \{\perp, 23, \perp\} \rangle]$$

and

$$v_2 = [\rangle \{1,2,3\}, \{1,2,3\}, \{1,2,3\}, \{2\} \rangle, \langle \{\text{"push(21)"}, \text{"push(22)"}, \text{"push(23)"}\}, \{\text{"pop"}, \text{"pop"}, \text{"pop"}\}, \{3,2,1\}, \{\perp, 21, \perp\} \rangle]$$

(which correspond to two different schedulings of the same input values) are input-equal.

5. Characterization for the shared memory model. In this section we state and prove the characterization theorem for t-resilient interactive$_k$ tasks in the shared memory model. The following claim will be useful in the proof of the main theorem for this model.

DEFINITION 10. *We say that an algorithm A is an* immediate-input *algorithm if the first step of every processor after writing an output in its private output register (as long as this is not the last (kth) output) is reading the next input from its private input register.*

CLAIM 1. If a distributed interactive$_k$ task, T, is solvable by a t-resilient algorithm, A, in the shared memory model, then T is also solvable by a t-resilient immediate-input algorithm, A'.

Proof. We show how to construct such an immediate-input algorithm, A', on the basis of the given algorithm, A. The algorithm A' will have all the shared registers of A plus an array of k Boolean multireader single-writer registers per processor (one bit for every input in T). The bits in all n arrays are initialized to "false." The state of a processor will consist of a pair (s_A, s_{add}), where s_A is a state of the processor in A and s_{add} reflects the additional parts, performed by A'. The algorithm A' will simulate algorithm A (changing s_A according to algorithm A) except at the following points:

- After a processor, P, writes the ℓth output ($\ell < k$), P's next step is to read the next, $(\ell+1)$st, input. However, P will leave the indicator corresponding to the $(\ell+1)$st input with the value "false," meaning that it did not want to read its input yet (in A). This will change only s_{add}, not s_A.
- When P is supposed to read its input (according to algorithm A) it will, instead, change the indicator corresponding to this input to "true." It will also change s_A according to algorithm A (and the value of this input).

It is not hard to verify that if A is t-resilient, then so is A'. □

We now define the notion of a t-founded node relative to a task T. Intuitively, a t-founded node is a "good" node from which a t-resilient algorithm can progress to correct outputs, regardless of the scheduling and of future revealed inputs. The definition of t-foundedness will be recursive.

DEFINITION 11. *Let*

$$v = [\langle A_1, \ldots, A_k, B_1, \ldots, B_k \rangle, \langle I^1_{A_1}, \ldots, I^k_{A_k}, O^1_{B_1}, \ldots, O^k_{B_k} \rangle]$$

be a node in the DAG. The node v will be called t-founded *relative to the interactive$_k$ task T if*

- *the node v is a "completed leaf"* $v = [\langle \{1,\dots,n\}_1, \dots, \{1,\dots,n\}_{2k}\rangle, \langle I^1,\dots,I^k,O^1,\dots,O^k\rangle]$ *and* $\langle I^1,\dots,I^k,O^1,\dots,O^k\rangle \in T$,

or

- *the node* $v = [\langle A_1, \dots, A_k, B_1, \dots, B_k\rangle, \langle I^1_{A_1}, \dots, I^k_{A_k}, O^1_{B_1}, \dots, O^k_{B_k}\rangle]$ *is not a completed leaf (namely* $B_k \neq \{1,\dots,n\}$*), and the following conditions hold:*
  1. *The node v has at least α non-t-equivalent sons, where* $\alpha = min\,(t+1, n-|B_k|)$.
  2. *For every possible t-subsequent, u, of v (u is not necessarily T-consistent), there exists an input-equal node u' that is a t-founded son of v.*

As we mentioned earlier, t-equivalent nodes have the same input indices revealed. Therefore, if two nodes are t-equivalent sons of some node, then the same processor made the t-advancement to both sons. Since the scheduler may fail up to t processors, t-resiliency necessitates, in the main phase, at least $t+1$ processors that can advance from any given situation. However, during the concluding phase there may be fewer than t active processors. (These are the processors that have not yet submitted their final output.) The number of processors able to make a t-advancement surely cannot exceed the number of active processors. This argument motivates the definition of α. If we require that there be at least α processors that are able to advance (i.e., at least α non-t-equivalent sons), some processor will always be able to advance from this state, no matter what the scheduling sequence is.

In addition, we need to have a legal consistent step available no matter what the input *values* are. This is embodied in the requirement for a t-founded input-equal son for every possible t-subsequent. Notice that the specific input values in a run cannot be chosen by the processors, while the output values *are* chosen by the processors. We remark that by the definition of t-foundedness, every path in the DAG which starts at the root, ends at some leaf, and proceeds only through t-founded nodes must end in a leaf belonging to T.

We are now in position to state and prove the characterization theorem for the shared memory model.

THEOREM 1. *In the terminating shared memory model, for $0 \leq t \leq n-1$, an interactive$_k$ task T is t-resilient iff the root of its DAG is t-founded relative to T.*

Proof. First we prove the $\Rightarrow$ direction: if the root of the DAG, $R_0 = [\langle \phi,\dots,\phi, \phi,\dots,\phi\rangle, \langle I^1_\phi, \dots, I^k_\phi, O^1_\phi, \dots, O^k_\phi\rangle]$ is *non-t*-founded relative to T, then T is not t-resilient. A high-level description of the proof is as follows: Assume that the root of the DAG is non-t-founded and yet there exists a t-resilient algorithm for the task T. We will show how the scheduler can force the system to make transitions along a sequence, s, of non-t-founded nodes in the DAG. Each node in s reflects the input values that were read and the output values that were produced by the processors. In every transition, at least one new input is read or one new output is produced. Since the number of inputs and outputs is finite, $2nk$, the number of such transitions will also be finite. This process will terminate either at a leaf of the DAG that is non-t-founded or in a state that is not represented by a node in the DAG. In the latter case, the inputs and outputs represent an inconsistent state. In either case, the scheduler has forced the algorithm to err.

A processor that had either terminated or read an input for which it has not yet given an output will be called a *nonreading* processor. Let v be a node in the sequence s. The scheduler's strategy will preserve two invariants regarding v:

1. The node v is non-t-founded, and every node in the DAG, u, that is input-equal to v is either inconsistent or is both consistent and non-t-founded. Notice that this invariant holds at the root, as we assume that the root is non-t-founded and there are no other nodes that are input-equal to the root.
2. The node, v, belongs to one of the following two categories:
   (a) The state represented by v is in the main phase and the number of nonreading processors in this state is smaller than $n - t$. (Notice that this invariant holds at the root, as there are no nonreading processors at the root.)
   (b) The state represented by v is in the concluding phase and at least $n - t$ processors have terminated (given their final output).

The scheduler will allow the system to advance by making a t-advancement while keeping these invariants. We will denote them as advancements from node R_ℓ to node $R_{\ell+1}$ ($\ell = 0$ at the root). In such an advancement there are two possibilities.

1. In the main phase, a t-advancement corresponds to an additional input and possibly several additional outputs.
2. In the concluding phase an advancement can be either an additional single input (only) or a single additional output.

We now present the detailed proof. Suppose, by way of contradiction, that T *is* t-resilient, namely, there exists a t-resilient algorithm, A, which implements T. By Claim 1 we can assume that this algorithm is an immediate-input algorithm. The root of the DAG corresponding to T, R_0, is *non-t*-founded. By Definition 11 there are two possible reasons for a node R_ℓ to be non-t-founded:

(A) R_ℓ has less than α non-t-equivalent sons.
(B) R_ℓ has a t-subsequent, $R'_{\ell+1}$, such that for every $R_{\ell+1}$ which is input-equal to $R'_{\ell+1}$ and is a son of R_ℓ in the DAG, $R_{\ell+1}$ is *non-t*-founded.

We describe an adversary scheduler which, as long as the system is at a node that is non-t-founded due to possibility (B), will force the algorithm to advance by choosing one of the non-t-founded input-equal sons. Notice that when the system moves from R_ℓ to $R_{\ell+1}$, the first invariant is maintained. ($R_{\ell+1}$ is non-t-founded and all of it's input-equals are either inconsistent (not a node in the DAG) or consistent (in the DAG) but non-t-founded.)

Let P_i be the processor associated with the t-advancement from R_ℓ to $R'_{\ell+1}$ (Definition 6). If the system is in the main phase, this advancement corresponds to P_i reading an input. If the system is in the concluding phase, this advancement corresponds to either P_i reading an input or P_i writing an output (case 2 of Definition 6).

In case P_i's step is reading an input, in the concluding phase, the adversary scheduler will activate P_i once and P_i will read this input, since A is an immediate-input algorithm. The resulting configuration will be $R_{\ell+1}$. In case P_i's step is reading an input in the main phase, the scheduler activates P_i (which reads its input) as above, but then the scheduler may perform additional activations according to the following cases.

If $n-t$ or more inputs of the final stage are known, then the scheduler will activate the $n - t$ processors that have read their final input (say, in round-robin order) until *all* of them give an output and terminate. By Definition 1, within a finite expected number of steps all $n-t$ processors will give outputs and terminate. The configuration reached after all $n - t$ processors have given their final output is $R_{\ell+1}$. Notice that this case preserves invariant 2(b), as we reach the concluding phase not only with

$n-t$ inputs of the final stage but also with $n-t$ outputs and terminated processors.

If, however, there are fewer than $n-t$ inputs of the final stage, then we know that the system remains in the main phase. In this case, let S_p denote the set of nonreading processors (after the read step of P_i). Notice that by invariant 2(a) it holds that $|S_p| \leq n-t$ because before P_i read its input, there were fewer than $n-t$ nonreading processors. The scheduler will do one of the following:

1. If $|S_p| < n-t$, then the resulting configuration is $R_{\ell+1}$ (no additional activations). In this case, the second invariant holds ($R_{\ell+1}$ is in the main phase and $|S_p| < n-t$).
2. If $|S_p| = n-t$, then the scheduler will activate the processors in S_p (say, in round-robin order) until exactly one of them gives an output. By Definition 1, such an output will be given within a finite expected number of steps. The resulting configuration will be $R_{\ell+1}$. This configuration adheres to the second invariant as now there are only $n-t-1$ nonreading processors.

When examining $R_{\ell+1}$ we see there are now two possibilities. The first is that the outputs given were inconsistent, in which case the algorithm has erred. The second possibility is that $R_{\ell+1}$ is consistent. The configurations $R_{\ell+1}$ and $R'_{\ell+1}$ are input equal. $R_{\ell+1}$ is a son of R_ℓ in the DAG. Since we assumed R_ℓ is non-t-founded due to (B), this implies that $R_{\ell+1}$ is non-t-founded.

The other t-advancement possible is that P_i's step in advancing from R_ℓ to $R'_{\ell+1}$ is writing an output. In this case the system must be in the concluding phase, and due to the second invariant we know that $n-t$ processors have already terminated. By Definition 1, since A is assumed to be a t-resilient algorithm, P_i will write an output, with probability 1, after being scheduled a finite number of times. Thus, the scheduler will activate P_i until it writes an output. In this case, the resulting configuration is the desired $R_{\ell+1}$. Again, if $R_{\ell+1}$ is inconsistent the algorithm has erred and otherwise, $R_{\ell+1}$ is non-t-founded (since all the t-advancements made by P_i in this case are input equal and thus non-t-founded). As the system was in the concluding phase and the second invariant was true before the advancement, it will also hold after the advancement.

Hence, in all cases, within a finite expected number of steps the scheduler can force the system to reach $R_{\ell+1}$ without failing any processor. The system is now at a non-t-founded node, $R_{\ell+1}$. The argument used for R_ℓ applies to $R_{\ell+1}$ as well, and can therefore be repeated.

Since the depth of the DAG is finite, after a finite number of such t-advancements we will reach a node R_f that is non-t-founded due either to possibility (A) or to the fact that it is a complete leaf which is not in T ($\langle I^1, \ldots, I^k, O^1, \ldots, O^k \rangle \notin T$).

In the main phase, a node v cannot be non-t-founded due to possibility (A): from the second invariant, less than $n-t$ processors are nonreading. Therefore, at least $t+1$ processors can read an additional input and thus v must have at least α non-t-equivalent sons. (Remember that $\alpha \leq t+1$ and that due to Definition 4 an additional input preserves consistency.)

If R_f is a completed leaf not in T, then the algorithm has erred, contradicting the assumption that algorithm A implements T without erring. Otherwise, R_f is non-t-founded due to possibility (A). This means that R_f has less than α non-t-equivalent sons. As $\alpha \leq t+1$, we know that at most t processors can be associated with a (consistent) t-advancement from R_f (Definitions 11 and 7). Denote the set of these processors by C_f.

For every output that is consistent for R_f (Definition 4) there exists a consistent

son of R_f which includes this output. Therefore, in the concluding phase when it holds that $n - t$ processors have already terminated (due to the second invariant), the fact that there are fewer than α processors able to make a t-advancement implies that the number of these processors is smaller than the number of active processors ($\alpha = min\,(t+1\ ,\ n - |B_k|)$, Definition 11). It follows that there exists at least one additional active processor (not in C_f) which cannot give a consistent output. Our adversary scheduler will activate all the processors not in C_f. Since the algorithm is t-resilient, one of the remaining (activated) processors must give an output within a finite expected number of steps. However, this processor's output is not consistent (the processor is not in C_f). This means that with this output, the system is at an inconsistent state and therefore (by Definition 4) there is an input that the scheduler can give the system for which it will err.

We have shown that the scheduler can force the system to advance along a sequence of non-t-founded nodes, and hence we have arrived at a contradiction in every possible case. This proves the $\Rightarrow$ direction of the theorem.

Now we will prove the $\Leftarrow$ direction; namely, if the root of the DAG corresponding to T is t-founded, then T is a t-resilient task. This is proven by presenting a generic t-resilient algorithm which implements T. In this algorithm, every processor will publish (in a shared register that the rest of the processors can read) its input as soon as it reads it and will also publish the output that it has chosen just before writing it in the output register. The frame of this algorithm will be a walk along a path in the DAG starting at its root, proceeding according to the inputs revealed and the produced outputs, and ending at a legal leaf that represents a full $2k$-tuple in T. This walk is executed commonly by all processors, and the way to achieve this is by applying consensus to every move.

There is a certain difference between the use of the DAG in this direction and its use in the previous direction. While in the first direction the current node represented the system's current state, in this direction the node represents a state the system "aspires" to. If a certain node, v, was agreed upon, it means that v represents input values that were published. However, not all the output values in v have necessarily been decided upon (and published) because they could belong to dormant processors. Such outputs will be decided upon and published by the respective processors when they are activated.

The algorithm we present uses as a subroutine an *extended consensus* protocol. This protocol allows consensus among processors when some of the participating processors do not offer a value but rather adopt one of the values offered by other processors. Processors that do not suggest values for the extended consensus protocol are called *passive processors* while processors that do offer a value for the extended consensus protocol are called *agile processors*. Given an $n-1$ resilient consensus protocol, and allowing β passive processors, we can build a $n-1-\beta$-resilient extended consensus protocol as follows:

- An agile processor will publish its offered value in a shared register and then proceed to execute the wait-free consensus protocol.
- A passive processor will wait until some agile processor suggests a value, adopt that value as its own, and then join the execution of the wait-free consensus protocol.

The algorithm will be carried out in "virtual rounds." In round $r+1$ each processor starts with node v_r, the t-founded node (in the DAG) that was agreed upon in round r (via consensus). Each processor then proceeds to pick, according to the DAG, a

son of v_r representing a $2k$-tuple, S_{i+1}. This node, u_r, is a t-subsequent of v_r, is t-founded, and includes the processor's new input, if such an input is available (i.e., P_i has not yet published its jth input even though it has given its $(j-1)$st output). If the system is in the concluding phase and no such node, u_r, exists, the processor P will look for a t-founded son of v_r that includes an additional output for P. The processor proposes the node it has found to the rest of the processors to agree upon. Then, the processors run consensus to agree on one of the proposed options. Since the option includes outputs (not necessarily outputs of the originating processor, i.e., the one whose option was chosen), each processor now checks the agreed node to see if it is required to output a value. Each of the relevant processors will output its required value immediately after receiving the result, v_{r+1}, of the $(r+1)$st consensus. The processor then proceeds to the next virtual round. Formally the algorithm for processor P_i is as follows:

INIT:

$v_r \leftarrow$ root(DAG). /* v_r is the current node in virtual round number r */
$a \leftarrow 1$ /* a holds P_i's current stage */
$r \leftarrow 1$ /* round counter */
$in_i \leftarrow input$

VIRTUAL ROUNDS:

do while $a \leq k$
denote v_r's components by $< I^1_{A_1}, \ldots, I^k_{A_k}, O^1_{B_1}, \ldots, O^k_{B_k} >= v_r$
if there exists $u_r =< I^1_{A'_1}, \ldots, I^k_{A'_k}, O^1_{B'_1}, \ldots, O^k_{B'_k} >$ **s.t.**
u_r is a son of v_r **and** u_r is t-founded
and
(either
$I^a_{A'_a} = I^a_{A_a} \cup \{in_i\}$ /* main or concluding phase - in_i is P_i's input */
and $A'_j = A_j \;\; j \neq a$
and $B_j \subseteq B'_j \;\; \forall j$
or
$B'_a = B_a \cup \{i\}$ /* concluding phase - only another output */
and $B'_j = B_j \;\; j \neq a$
and $A'_j = A_j \;\; \forall j$
)
then $v_{r+1} \leftarrow$ *extended-consensus*(u_r, r)
else $v_{r+1} \leftarrow$ *extended-consensus*$(passive, r)$
if $i \in B'_a$ /* an additional output of P_i was decided upon */
then
output$(O^a_{\{i\}})$ /* output the corresponding value */
if $a = k$
then terminate
$a \leftarrow a + 1$
$in_i \leftarrow input$
$r \leftarrow r + 1$
end /* of while */

The different consensus rounds are separate and use separate sets of registers. We will use a multivalue consensus protocol with the addition of "passive inputs"

(belonging to passive processors). The requirement from the protocol is that its output will be the input of one of the agile processors. Any known wait-free protocol for consensus (e.g., [1, 2, 9, 26]) can easily be modified to comply with this requirement. Notice that if up to β processors can be passive in a given round, then the modified consensus protocol is $(n-\beta-1)$-resilient. In the concluding phase, it is possible that the processors that have missing inputs are faulty, and so every remaining processor must be able to advance on its own (without waiting for values offered by other processors).

Let us now prove that the given algorithm is correct and t-resilient. According to the remark from the end of section 4, following a t-founded path leads to a leaf in T. This means (by the definition of a t-founded leaf in Definition 11) that the outputs given by the system match the inputs it received, according to the task T. Hence the algorithm always gives correct outputs for any legal input. As every call to the consensus subroutine takes bounded expected time, and the other operations are finite, within bounded expected time the algorithm will terminate. Therefore the algorithm is correct.

To show t-resiliency we will show that the system can always advance (even if t processors have failed, as long as some processor is active), i.e., there will always be an agile processor that can find a u_r as required. Notice that v_r is always t-founded (the root is t-founded by the assumption, and only t-founded nodes are chosen by the algorithm during the execution). Also, for a given node, the number of passive processors is less than $n-\alpha$ as at least α processors can find a u_r as required. Now let us show that there are indeed enough agile processors. We will do this for two different cases, according to the phase that the system is in:

- In the main phase and in the concluding phase when less than $n-t$ processors have terminated, $\alpha = t+1$. Therefore, even if t processors have failed, there exists at least one additional processor that can make a t-advancement from this t-founded node. This processor will be agile in the consensus round for v_r and thus the extended consensus will be computed (in bounded expected time) in this round.
- In the concluding phase when $n-t$ processors have already terminated, α equals the number of not-yet-terminated processors. Therefore, every processor that has not yet terminated and will be activated can find a u_r as required. Such a processor will be agile in the respective extended consensus round.

Altogether, we have shown that for every node on the path there are enough processors that can make a t-advancement from it. Thus, the algorithm is t-resilient, as claimed. □

6. Characterization for message passing. In this section we will examine the resiliency of interactive tasks in the message passing model, where it is possible to perform t-resilient consensus only for $t \leq \lfloor \frac{n-1}{2} \rfloor$. In the terminating message passing model a processor terminates after giving its final output. After $n-t$ processors have terminated, the remaining t processors could be disconnected from each other by the scheduler, and thus will have no way of "coordinating" amongst themselves. The disconnection is possible since all t processors might be faulty and so no processor can wait for a message from another processor. (Recall that in an asynchronous system the processors have no way of distinguishing between a slow and a faulty processor.) Each of the t slow processors will know what the fast $n-t$ processors have decided but will have no way of knowing what any one of the other slow processors has decided. In order for a task to be t-resilient, the slow processors must be able to decide on their

value in a "consistent" manner, each without knowing the others' decisions. Notice that a particularly slow processor may be at the beginning of its work, i.e., it has not yet read even its first input. Such a processor must be able to function correctly through *all* the stages of the task. Following these considerations, we focus on nodes in the DAG, $D(T)$, that have at least $n-t$ outputs of the final stage revealed. We call these nodes *semiterminal* nodes. In these nodes each additional input *or* output is considered a t-advancement (by Definition 6). Notice that for semiterminal nodes we require at least $n-t$ *outputs* of the final stage, while in the concluding phase we required only $n-t$ *inputs* of the final stage.

DEFINITION 12. *A tree R_i that is a subgraph of $D(T)$ will be called* terminal *for P_i if the following conditions hold:*

- *The root of R_i, $v=[S,Q]$ where $S=\langle A_1\ ,\ \ldots\ ,\ A_k\ ,\ B_1\ ,\ \ldots\ ,\ B_k\rangle$ and $Q=\langle I^1_{A_1},\ \ldots,\ I^k_{A_k},\ O^1_{B_1},\ \ldots,\ O^k_{B_k}\rangle$, is a semiterminal node ($|B_k|\geq n-t$) such that $i\notin B_k$. (Processor P_i has not yet terminated.)*
- *Let ℓ denote the last stage for which P_i gave an output. When we look at the inputs and outputs for processor P_i, the following condition holds: For every input for the $(\ell+1)$st stage there is an output for the $(\ell+1)$st stage such that for every input for the $(\ell+2)$nd stage there is ... for every input for the kth stage there is an output for the kth stage such that the t-advancements corresponding to these inputs and outputs form a path in R_i. In the case that the $(\ell+1)$st input is already given in v, then the condition starts with the existence of an output for the $(\ell+1)$st stage such that for every input for the $(\ell+2)$nd stage, etc.*

Intuitively, the tree R_i represents a possible strategy for P_i to react to every possible sequence of inputs presented to it, when the system is at the state represented by the root, v. The next definition captures the requirement that terminal trees of different processors should be compatible.

DEFINITION 13. *Let v be a semiterminal node and let $\{i_1,\ldots,i_j\}=\{1,\ldots,n\}\backslash B_k$ (i.e., the set of processors that have not yet terminated in v). Let $R_{i_1},\ldots,R_{i_j}$ be j terminal trees for processors $P_{i_1},\ldots,P_{i_j}$, respectively. These trees are called T-compatible if the following conditions hold:*

- *$R_{i_1},\ldots,R_{i_j}$ have the same semiterminal root v:*
  $v=[\langle A_1\ ,\ \ldots\ ,\ A_k\ ,\ B_1\ ,\ \ldots\ ,\ B_k\rangle,\langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle]$.
- *For every $\langle I^1,\ldots,I^k\rangle\in IN(T)$ which is an extension of $\langle I^1_{A_1},\ldots,I^k_{A_k}\rangle$, the outputs produced along the set of paths (one path per tree) that correspond to this input give a full output vector $\langle O^1,\ldots,O^k\rangle$ such that $\langle I^1,\ldots,I^k,\ O^1,\ldots,O^k\rangle\in T$ and $\langle O^1,\ldots,O^k\rangle$ is an extension of $\langle O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle$.*

We use these definitions to describe our "building block," t-validity. It will play the role that t-foundedness played in the shared memory model (section 5).

DEFINITION 14. *Let*

$$v=[\langle A_1\ ,\ \ldots\ ,\ A_k\ ,\ B_1\ ,\ \ldots\ ,\ B_k\rangle,\langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle]$$

be a node in the DAG $D(T)$, and let $\alpha=min\ (t+1\ ,\ n-|B_k|)$. We say that v is t-valid *relative to the task T when*

- *The node v is a leaf ($v=[\langle\{1,\ldots,n\}_1,\ \ldots,\ \{1,\ldots,n\}_{2k}\rangle,\ \langle I^1,\ldots,I^k,\ O^1,\ldots,O^k\rangle]$), and $\langle I^1,\ldots,I^k,O^1,\ldots,O^k\rangle\in T$;*

or

- *The node v is an internal node (not a leaf) and the following conditions hold:*
  1. *v has at least α non-t-equivalent sons.*

2. *For every t-subsequent, u, of v there exists an input-equal node u' that is a t-valid son of v.*
3. *If v is semiterminal, then for every $i \notin B_k$ there exists a terminal tree, R_i, for P_i, such that the trees $\{R_i\}_{i \notin B_k}$ are T-compatible. We will call one of these sets, say, the first in canonical order that complies with the above condition, the* chosen set *for v, denoted by $ch(v)$.*

The characterization theorem is formulated in the same way as Theorem 1, but the different definitions of t-foundedness versus t-validity make the difference between the shared memory and the message passing models. The additional requirement (3) explains why certain tasks, solvable in the shared memory model, cannot be solved in the message passing model.

THEOREM 2. *Consider the terminating message passing model, and let t satisfy $t \leq \lfloor \frac{n-1}{2} \rfloor$. An interactive$_k$ task, T, is t-resilient iff the root of its corresponding DAG is t-valid relative to T.*

Proof. The proof follows the proof of Theorem 1 closely. The differences between these proofs are due to the nature of the communication model. For the $\Rightarrow$ direction, we use an algorithm which, starting at a t-valid node, proceeds along a path of t-valid nodes until a semiterminal node is reached. The algorithm uses an *extended consensus* subroutine (see proof of Theorem 1) which guarantees that all the processors agree on the same nodes along the path. In the message passing model, the resiliency of the extended consensus protocol is $min(\lfloor \frac{n-1}{2} \rfloor, n-1-\beta)$, where β is an upper bound on the number of passive processors (processors that take part in the consensus protocol but do not suggest a value). In our algorithm, no processor terminates before a semiterminal node is reached. That is, no processor writes its final (kth) output before a node which includes the final outputs of at least $n-t$ processors has been agreed upon. This precaution guarantees that at least $n-t$ processors remain active throughout the main phase of the algorithm. This number is large enough to enable each round of the extended consensus algorithm to terminate, as there is at least one agile processor in each round, and at least $\lfloor \frac{n+1}{2} \rfloor$ processors take part in the sequence of consensus executions until a semiterminal node is reached and agreed upon.

We associate with every t-valid semiterminal node, v, its chosen set, $ch(v)$ (the existence of this set follows from Definition 14). Once the system reaches and agrees upon such a semiterminal node, all processors P_i with $i \in B_k$ can produce their final output and terminate. Each remaining processor can now produce a local output as a response to every new local input it receives by proceeding along the corresponding path in its tree. This requires no communication and coordination with the remaining processors. The compatibility of the terminal trees guarantees that this process leads to a leaf satisfying the input-output relations of the task T. In order to simplify the algorithm, we will use a pruned DAG, denoted by $D'(T)$. This DAG will be the same as D(T) for all nodes that are not semiterminal. For semiterminal nodes, $D'(T)$ will include only their chosen set. That is, for semiterminal node v, $D'(T)$ will include only $ch(v)$ (and not other alternative advancements from v). The algorithm will run on $D'(T)$ rather than $D(T)$.

Formally, the algorithm for P_i is as follows.

INIT:
$v_r \leftarrow \text{root}(D'(T))$. /* v_r is the current node in virtual round number r */
$a \leftarrow 1$ /* a holds P_i's current stage */
$r \leftarrow 1$ /* round counter */
$in_i \leftarrow input$

VIRTUAL ROUNDS:
do while $a \leq k$
denote v_r's components by $< I^1_{A_1}, \ldots, I^k_{A_k}, O^1_{B_1}, \ldots, O^k_{B_k} >= v_r$
if there exists $u_r =< I^1_{A'_1}, \ldots, I^k_{A'_k}, O^1_{B'_1}, \ldots, O^k_{B'_k} >$ **s.t.**
u_r is a son of v_r **and** u_r is t-valid
and
(either
$I^a_{A'_a} = I^a_{A_a} \cup \{in_i\}$ /* main phase - in_i is the new input */
and $A'_j = A_j$ for all j, $j \neq a$
and $B_j \subseteq B'_j$ for all j
or
$B'_a = B_a \cup \{i\}$ /* concluding phase - only another output */
and $B'_j = B_j$ for all j, $j \neq a$
and $A'_j = A_j$ for all j
)
then if $|B_k| < n - t$/* consensus is possible until $n - t$ processors terminate */
then $v_{r+1} \leftarrow extended - consensus(u_r, r)$
else $v_{r+1} \leftarrow u_r$ /* no consensus - proceed along terminal tree */
else $v_{r+1} \leftarrow extended - consensus(passive, r)$ /* this case is only possible in the main phase */
if $i \in B'_a$ /* an additional output of P_i was decided upon */
then (**if** $a < k$
then
output$(O^a_{\{i\}})$ /* output the corresponding value */
$a \leftarrow a + 1$
$in_i \leftarrow input$
elseif $|B'_k| \geq n - t$
then
output$(O^a_{\{i\}})$ /* output the corresponding value */
terminate
$r \leftarrow r + 1$
end /* of while */

The consensus protocol used by the extended consensus must be $\lfloor \frac{n-1}{2} \rfloor$-resilient and terminating. The different consensus rounds are separate and use separate message numbers. (Any consensus protocol that complies with these requirements can be used as a subroutine. Examples of such consensus protocols are [7, 6]). Notice that passive processors are counted for the number of processors participating in the consensus as they can "help out." However, at least one agile processor must participate in the consensus (to suggest the next value). There will always be such a processor for the same reasons as in the shared memory model (in the main phase at least $\alpha = t+1$ processors can find the required u_r as the current node is always t-valid, and in the concluding phase there is no need for consensus).

The $\Leftarrow$ direction is also similar to the shared memory case. As in Claim 1, we can assume that the algorithms used are immediate input algorithms. (The proof for the message passing model is identical to that of the shared memory case.) We present a scheduler that forces the system to always remain in a non-t-valid node. By Definition 14, the possible cases at node v (which is non-t-valid) are

0. v is a non-t-valid leaf;
1. v has less than α non-t-equivalent sons;
2. v has a t-subsequent, v_1, such that for every u' that is input-equal to v_1 and is a son of v, it holds that u' is *non-t*-valid;
3. v is semiterminal and one of the following is true:
   (a) There exists $i \notin B_k$ for which there is no terminal tree rooted at v.
   (b) Every collection of terminal trees $\{R_i\}_{i \notin B_k}$ rooted at v is not compatible.

As in the shared memory model, let us assume toward a contradiction that there exists a t-resilient immediate-input algorithm for T. We will start at the root of the DAG and the processors will follow this algorithm. As long as the node representing the system's current state, v, is non-t-valid due to possibility 2, the scheduler forces the processors to advance to a state represented by u' that is input-equal to v_1. This will continue until we reach a node v_f that is non-t-valid either due to possibility 1 or possibility 3 or because v_f is a non-t-valid leaf. If v_f is non-t-valid due to possibility 1 or is a leaf, then for exactly the same reasons as in the proof for the shared memory model we have shown a contradiction.

It remains to show that the other alternative (non-t-validity due to possibility 3) also leads to contradiction. The first case is when there exists an i for which there is no terminal tree. By Definition 12, this means that there exists a strategy for the scheduler (supplying inputs) such that every strategy for the processor P_i leads to a partial $2k$-tuple which is not a partial vector of any full $2k$-tuple in T. This contradicts the correctness of the algorithm.

The second case is when each selection of terminal trees is not compatible (i.e., every set of terminal trees, one for each active processor, is incompatible). In this case the scheduler activates these active processors and withholds all messages sent between them. Now each one must act on its own. Let us now show that there is a strategy for the scheduler which causes the processors to err with positive probability. In order to do that we will construct one set of terminal trees for all the remaining active processors, using a "roll-back" technique. The construction starts with the system at the state in which it arrived at the semiterminal node v. Now for every processor P_i where $i \notin B_k$, the tree T_i is built recursively, as follows: For every node, u, in the tree (starting the recursion at v), the scheduler determines the son by presenting the processor P_i with a legal input at u and the grandson by activating P_i until it produces an output (which must happen within bounded time, by the requirements of t-resilient algorithms). The process is repeated in the next recursion level, with respect to the grandson. To determine the continuation from u with respect to other inputs, the scheduler "rolls back" the system to the same state it was in u and then supplies another input. This process is done with respect to every legal input for P_i at u. The end of the recursion along each path is when P_i supplies its last (kth) output.

This process builds a set of terminal trees. Every branch along each tree represents a strategy for the corresponding processor, which is used with positive probability. Probabilities of different processors are independent, since their random inputs are independent, and the processors cannot communicate. By our assumption, the terminal trees are incompatible. Therefore, there exist input sequences (one per processor) such that the output sequences resulting by following the corresponding trees yield a full input-output $2k$-tuple that is not in T. The scheduler will supply these inputs and then the processors will give incorrect answers (due to the definition of incompatible terminal trees) with positive probability. This contradicts the assumption that the

algorithm implements T with no error. Therefore the condition in the theorem is necessary. □

7. Concluding remarks. Denote by REF the class of randomized error-free protocols and by DC the class of deterministic protocols with access to consensus. Since consensus has wait-free REF solution, it follows that any finite interactive task solvable in the DC model is also solvable in the REF model. The proof of Theorems 1 (for the shared memory model) and 2 (for the message passing model) implies that any finite interactive task solvable in the REF model is also solvable in the DC model. Stated formally, we have the following theorem.

THEOREM 3. *Consider the following two families of protocols:*

- *deterministic protocols with access to consensus (DC),*
- *randomized error-free protocols (REF).*

An interactive$_k$ task T is t-resilient in the DC family iff it is t-resilient in the REF family, where the range of t is

- $0 \leq t \leq n-1$ *in the terminating shared memory model,*
- $0 \leq t \leq \lfloor \frac{n-1}{2} \rfloor$ *in the terminating message passing model.*

Results concerning the structure of the "resiliency hierarchy" in both the shared memory and the message passing models can easily be inferred from our characterizations. They extend similar results for decision tasks, proven by Chor and Moscovici in [13]. Some of these implications are as follows:

- Shared memory is strictly more powerful than message passing for the same resiliency (except at the two lowest resiliencies): $IMP_t \subsetneq ISM_t$ $(1 < t < n)$, while $IMP_0 = ISM_0$ and $IMP_1 = ISM_1$.
- With respect to difference resiliencies, message passing and shared memory, namely IMP_t and ISM_{t+i}, are incompatible for $0 < i < n - t$. (The results in [13] already prove this.)
- In the range of resiliency where network partition is not possible ($t \leq \lfloor \frac{n-1}{2} \rfloor$), *nonterminating* protocols in the shared memory and message passing models have the same capabilities: namely, nonterminating IMP_t = nonterminating $ISM_t = ISM_t$. For t in the range $\frac{n}{2} \leq t \leq n-1$, nonterminating $ISM_t = ISM_t$.

7.1. Related work. Bar-Noy and Dolev [6], and consequently Attiya, Bar-Noy and Dolev [4], have investigated emulation strategies of shared memory in nonterminating message passing systems. One consequence of their work is that $ISM_{n-1} \subseteq$ nonterminating $IMP_{\lfloor \frac{n-1}{2} \rfloor}$. In fact, their construction yields the stronger result $ISM_{\lfloor \frac{n-1}{2} \rfloor} \subseteq$ nonterminating $IMP_{\lfloor \frac{n-1}{2} \rfloor}$. However, no characterization can be derived from their techniques. Also, it is not clear if their "local" approach can be extended to yield the equality between these classes (which does follow from our characterization).

Plotkin [23] and Herlihy [17] have studied the implementation of concurrent objects in the shared memory model. They have shown that every sequential system has a concurrent wait-free implementation, given access to a wait-free consensus subroutine. We note that our results give a natural way to implement finite versions of concurrent objects, and so they shed light on the use of randomization for wait-free concurrent objects. In fact, the concurrent objects in [17] refer to $t = n-1$ (wait-free objects) while interactive tasks can implement a wider range of fault-tolerance.

Taubenfeld, Katz, and Moran [28] and Taubenfeld and Moran [29] have studied a weaker type of crash failures—initial faults. In the initial faults model, each processor

either is initially crashed or remains active forever. Taubenfeld, Katz, and Moran found necessary and sufficient conditions for solvability of distributed *decision* tasks with respect to deterministic protocols in the initial fault message passing and shared memory models. Surprisingly, the characterization is the same as the one for general crash faults using randomized protocols [13]. Indeed, these equalities no longer remain valid when interactive, rather than decision, tasks are considered. This is due to the scheduler's limited powers in the initial faults model. Therefore, it is not surprising that there are interactive tasks that are t-resilient in the initial fault model but are not t-resilient for general crash faults. As an example, consider the following interactive$_2$ task: Inputs for both stages are all binary n-vectors. The value in each component of the first output equals the sum of the first inputs over some subset S of size $n - t$ (the subset is not predetermined). The value in each component of the second output equals the sum of the second inputs over the *same* subset S. It is easily seen that this task is resilient to t *initial* faults but is not in ISM_t since in ISM_t the scheduler can fail a previously active processor, thus leaving the second input unknown. This confirms the intuition that initial faults alone are not sufficient to represent a realistic fault model. (In the other direction, every task in ISM_t can be solved in a system with at most t crashes that can implement a consensus subroutine. Since consensus is solvable deterministically in the initial faults model, this implies that ISM_t is strictly contained in the class of t initial faults.)

The modular way in which consensus is used implies that our characterization is also a sufficient condition for t-resilience in *deterministic* systems augmented with *stronger* mechanisms that make consensus possible, for example, the failure detectors of Chandra and Toueg [10]. However, the question whether our conditions are also necessary requires a closer look. For example, suppose one has at his possession an accurate and reliable failure detector. If the detector announces that a processor has crashed, then we are guaranteed this processor will not become active later. In such a case we may be able to solve a natural modification of, for example, the parity task. We assign $\perp$ as the input of processors that crashed before supplying an input, and the $\perp$ values do not influence the output. Under such conditions, the parity is no longer unsolvable in the presence of one failure. This example implies that exact characterization of resilience in the presence of failure detectors strongly depends on the specific properties of the detector.

Despite substantial differences, there is a common feature to the initial faults model and the one with strong failure detectors. In both, the power of the adversary is severely restricted, further than what is "needed" to enable deterministic consensus. Any task which satisfies our characterization for t-resilience will also be t-resilient in these restricted adversary models. But whether the condition is also necessary crucially depends on the strength (or weakness) of the adversary.

Acknowledgments. Thanks to Hagit Attiya and Oded Goldriech for helpful discussions and suggestions and for their comments on earlier versions of this manuscript, to Moshi Molcho for his help and support, and to the anonymous referees for helpful comments.

REFERENCES

[1] J. Aspnes, *Time- and Space-Efficient Randomized Consensus*, J. Algorithms, 14 (1993), pp. 414–431.

[2] J. Aspnes and M. Herlihy, *Fast randomized consensus using shared memory*, J. Algorithms, 11 (1990), pp. 441–461.

[3] J. Aspnes and O. Waarts, *Randomized Consensus in Expected $O(n \log^2 n)$ Operations per Processor*, SIAM J. Comp., 25 (1996), pp. 1024–1044.
[4] H. Attiya, A. Bar-Noy, and D. Dolev, *Sharing memory robustly in message passing systems*, J. ACM, 42 (1995), pp. 124–142.
[5] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, *Renaming in an asynchronous environment*, J. ACM, 37 (1990), pp. 524–548.
[6] A. Bar-Noy and D. Dolev, *A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment*, Math. Systems Theory, 26 (1993), pp. 21–39.
[7] M. Ben-Or, *Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols*, in Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, 1983, pp. 27–30.
[8] E. Borowsky and E. Gafni, *Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations*, in Proceedings of the 25th Symposium on the Theory of Computing, 1993, pp. 91–100.
[9] G. Bracha and O. Rachman, *Randomized Consensus in Expected $O(n^2 \log n)$ Operations*, TR–671, Technion, Haifa, Israel, 1991.
[10] T. D. Chandra and S. Toueg, *Unreliable failure detectors for reliable distributed systems*, J. ACM, 43 (1996), pp. 225–267.
[11] B. Chor, A. Israeli, and M. Li, *Wait-Free Consensus Using Asynchronous Hardware*, SIAM J. Comput., 23 (1994), pp. 701–712; conference version in Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, 1987, pp. 86–97.
[12] B. Chor, M. Merritt, and D. Shmoys, *Simple constant time consensus protocols in realistic failure models*, J. ACM, 36 (1989), pp. 591–614.
[13] B. Chor and L. Moscovici, *Solvability in Asynchronous Environments*, in Proceedings of the 30th Symposium on Foundations of Computer Science, 1989, pp. 422–427.
[14] B. Chor and L. Nelson, *Resiliency of Interactive Distributed Tasks*, in Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, 1991, pp. 37–49.
[15] D. Dolev, Dwork C., and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, J. ACM, 34 (1987), pp. 77–97.
[16] M. Fischer, N. Lynch, and M. Paterson, *Impossibility of distriburted consensus with one faulty process*, J. ACM, 32 (1985), pp. 374–382.
[17] M. Herlihy, *Wait-free synchronization*, ACM Trans. Programming Languages Systems, 13 (1991), pp. 124–149.
[18] M. Herlihy and N. Shavit, *The Asynchronous Computability Theorem for t-Resilient Tasks*, in Proceedings of the 25th Symposium on the Theory of Computing, 1993, pp. 111–120.
[19] L. Lamport, *On interprocess communication*, Distrib. Comput., 1 (1986), pp. 77–101.
[20] D. Lehmann and M. Rabin, *On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem*, in Proceedings of the 8th Principles of Programming Languages, 1981, pp. 133–138.
[21] M. C. Loui and H. H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, in Advances in Computing Research, JAI press, Greenwich, CT, 1987, pp. 163–183.
[22] S. Moran and Y. Wolfstahl, *Extended impossibility results for asynchronous complete networks*, Inform. Process. Lett., 26 (1987), pp. 145–151.
[23] S. Plotkin, *Sticky Bits and the Universality of Consensus*, in Proceedings of the 8th ACM Symposium on Principles of Distributed Computing, 1989, pp. 159–176.
[24] M. Rabin, *Randomized Byzantine Generals*, in Proceedings of the 24th Symposium on Foundations of Computer Science, 1983, pp. 403-409.
[25] M. Rabin, *The choice coordination problem*, Acta Inform., 17 (1984), pp. 121–134.
[26] M. Saks, N. Shavit, and H. Woll, *Optimal Time Randomized Consensus—Making Resilient Algorithms Fast in Practice*, in Proceedings of the 2nd ACM Symposium on Discrete Algorithms, 1991, pp. 351–362.
[27] M. Saks and F. Zaharoglou, *Wait-Free k-set Agreement is Impossible: The Topology of Public Knowledge*, in Proceedings of 25th Symposium on the Theory of Computing, 1993, pp. 111–120.
[28] G. Taubenfeld, S. Katz, and S. Moran, *Initial failures in distributed computations*, Internat. J. Parallel Programming, 18 (1989), pp. 255–276.
[29] G. Taubenfeld and S. Moran, *Possibility and impossibility results in a shared memory environment*, Acta Inform., 33 (1996), pp. 1–20.