Impossibility Results in the Presence of Multiple Faulty Processes

(Preliminary Version)

Gadi Taubenfeld* Shumel Katz† Shlomo Moran‡

Abstract. We investigate the impossibility of solving certain problems in an unreliable distributed system where multiple processes may fail. We assume undetectable crash failures which means that a process may become faulty at any time during an execution and that no event can happen on a process after it fails. A sufficient condition is provided for the unsolvability of problems in the presence of multiple faulty processes. Several problems are shown to be solvable in the presence of $t-1$ faulty processes but not in the presence of t faulty processes for any t. These problems are variants of problems which are unsolvable in the presence of a single faulty process (such as consensus, choosing a leader, ranking, matching). In order to prove the impossibility result a contradiction is shown among a set of axioms which characterize any fault-tolerant protocol solving the problems we treat. In the course of the proof, we present two results that appear to be of independent interest: first, we show that for any protocol there is a computation in which some process is a *splitter*. This process can split the possible outputs of the protocol to two disjoint sets. In case that the protocol is also fault-tolerant, then this splitter must be a *decider*, that can split its own output values into two different singletons. These results generalize and expand known results for asynchronous systems.

1 Introduction

In this paper we investigate the possibility and impossibility of solving certain problems in an unreliable distributed system where a number of processes may fail. We assume undetectable crash failures which means that no event can happen on a process after it fails and that failures are undetectable. For any $1 \leq t < n$, where n is the number of processes, we define a class of problems which cannot be solved in a completely asynchronous system where t processes may fail. This implies a (necessary) condition for solving a problem in such an unreliable system. These results generalize previously known impossibility results for completely asynchronous systems, and prove new results.

*Computer Science Department Yale University, New Haven, CT 06520. Supported in part by the National Science Foundation under grant DCR-8405478, by the Hebrew Technical Institute scholarship, and by the Guttwirth Fellowship.

†Computer Science Department, Technion, Haifa 32000, Israel.

‡Computer Science Department, Technion, Haifa 32000, Israel. Supported in part by Technion V.P.R. Funds - Wellner Research Fund, and by the Foundation for Research in Electronics, Computers and Communications, administrated by the Israel Academy of Sciences and Humanities.

Various authors have investigated the nature of systems where only a *single* process may fail (i.e., $t = 1$). It is proven in [FLP] that in asynchronous systems there cannot exist a nontrivial consensus protocol that tolerates even a single process (crash) failure. This fundamental result has been extended to other models of computation [DDS,DLS]. Various extensions [MW, Ta, BMZ], also for a single fault, prove the impossibility of other problems using several new techniques. Other recent works point out some specific problems that can be solved in asynchronous systems with numerous faulty processes, and prove impossibility results for other problems [ABDKPR,BW,DDS]. In [TKM2] a necessary and sufficient condition is provided for solving problems in an unreliable asynchronous message passing systems where undetectable *initial* failures may occur. Initial failures are a very weak type of failures where it is assumed that processes may fail only prior to the execution. Results for asynchronous shared memory systems which support only atomic read and write operations, similar to those presented here appear in [TM].

Define an input vector to be a vector $\vec{a} = (a_1, ..., a_n)$, where a_i is the input value of process p_i. A crucial assumption in all the above results (for a single process failure) is that the set of input vectors is "large enough". To demonstrate this fact, consider the consensus problem where only two input vectors are possible: either all processes read as input the value "zero" or all processes read as input the value "one". It is easy to see that under this restriction, the problem can be solved assuming any number of process failures. One of the consequences of our result is to identify a property (or a *promise* [ESY]) which a set of input vectors should satisfy so that a problem can be solved in the presence of $t-1$ faulty processes but not in the presence of t faulty processes.

We show variants of the problems which are known to be unsolvable in the presence of a single faulty process (such as consensus, choosing a leader, ranking, matching, and sorting) and prove that the variants can be solved in the presence of $t-1$ faulty processes but not in the presence of t faulty processes for any t. An example is the consensus problem where the promise is that for each input vector, $|\#1 - \#0| \geq t$. (i.e., the absolute difference between the number of ones and the number of zeroes is at least t.)

The proof of our result is constructed as follows. We first identify a class of *protocols* that cannot tolerate the failure of t processes, when operating in a completely asynchronous system. Then, we identify those *problems* which force every protocol which solves them to belong to the above class of protocols. Hence, these problems cannot be solved in a completely asynchronous system where t processes may fail.

As in [FLP], we differentiate between a process having *reached* a decision, and a stage at which the eventual decision to be reached by a process is *uniquely determined* (but usually not yet known at a process). The class of protocols for which we prove the impossibility result is characterized by two requirements on the possible input and decision (output) values of each member in the class. For the input, it is required that (for each protocol) there exists a group of at least $n - t$ processes and there exist input values such that after all the $n - t$ processes in the group read these input values, the eventual decision value of at least one of them is still not uniquely determined. As for the decision values, the decision of different processes should have the following mutual dependency: the eventual decision value of any (single) process is uniquely determined as soon as all other processes decide.

In order to prove the above result for protocols, we use an axiomatic approach for proving properties of protocols (and problems) due to Chandy and Misra [CM1,CM2]. The idea is to capture the main features of the model and the features of the class of

protocols for which one wants to prove the result by a set of axioms, and to show that the result follows from the axioms. Unlike in [CM1], we define a model in which all messages are eventually delivered. We will present six axioms capturing the nature of asynchronous message passing systems, a single axiom expressing the fact that at most t processes may be faulty, and two axioms defining the class of protocols for which we want to prove the impossibility result. We then show that no protocol in the class can tolerate t faulty processes, by showing that the set of the nine axioms is inconsistent.

The rest of the paper is organized as follows. In Section 2 the notions of a problem and a protocol are defined. In Section 3 the properties of asynchronous message passing systems are stated. In Section 4 the notions of *decision* and *robustness* are introduced. In Section 5 we prove two results that appear to be of independent interest: first, we show that for any protocol there is a computation in which some process is a *splitter*. This process can split the possible outputs of the protocol to two disjoint sets. In case that the protocol is also fault-tolerant, then this splitter must be a *decider*, that can split its own output values into two different singletons. In Section 6, we use the result of the previous section along with a condition called dependency to show any protocol satisfying the conditions cannot tolerate t process failures. In Section 7, we finally identify the class of problems that cannot be solved in the presence of t process failures.

2 Definitions and Basic Notations

First, the type of problems we consider is described. Let I and D be sets of input values and decision (output) values, respectively. Let n be the number of processes, and let $\bar{I}$ and $\bar{D}$ be subsets of I^n and D^n, respectively. A problem T is a mapping $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ which maps each n-tuple in $\bar{I}$ to subsets of n-tuples in $\bar{D}$. We call the vectors $\vec{a} = (a_1, ..., a_n)$ where $\vec{a} \in \bar{I}$, and $\vec{d} = (d_1, ..., d_n)$ where $\vec{d} \in \bar{D}$, the *input* vector and *decision* vector, respectively. In that case, we say that a_i is the input value of process p_i, and d_i is the decision value of process p_i.

Following are some examples of problems, referred to later in the paper (the input vectors for all problems are from I^n for an arbitrary set I): (1) The *permutation* problem, where each process $p_i (i = 1..n)$ decides on a value v_i from D, $D = 1, ..., n$, and $i \neq j$ implies $v_i \neq v_j$; (2) The *consensus* problem, where all processes are to decide on the same value from an arbitrary set D; (3) The (leader) *election* problem, where exactly one process is to decide on a distinguished value from an arbitrary set D; (4) The *sorting* problem, where all processes have input values and each process p_i decides on a value identical to the i^th smallest input value; and (5) The *rotation* problem, where each process p_i decides on a value identical to the input value of the process $p_{i(mod\ n)+1}$.

A *protocol* is a nonempty set C of *computations* and a set of process id's (abbv. processes), $N = \{p_1, ..., p_n\}$. A computation is a *finite* sequence of events. There are four types of events: *send*, *receive*, *input*, and *decide*. A *send* event, denoted $([send, m, p_k], p_i)$, represents sending a message m to process p_k by process p_i. A *receive* event, denoted $([receive, m], p_k)$, represents receiving a message m by process p_k. An *input* event, denoted $([input, a], p_i)$, represents reading a value a by process p_i. A *decide* event, denoted $([decide, d], p_i)$, represents deciding on a value d by process p_i. We use the notation (e, p_i) to denote an arbitrary event, which may be an instance of any of the above types of events. For an event (e, p_i) we say that it occurred *on* process p_i. An event is *in* a

computation iff it is one of the events in the sequence which comprises the computation.

In the rest of this paper Q denotes a set of processes where $Q \subseteq N$. The symbols x, y, z denote computations. Also $\langle x; y \rangle$ is the sequence obtained by concatenating the two sequences x and y. An *extension* of a computation x is a computation of which x is a prefix. For an extension y of x, $(y - x)$ denotes the suffix of y obtained by removing x from y. For any x and p_i, let x_i be the subsequence of x containing all events in x which are on process p_i. Computation y *includes* x iff x_i is a prefix of y_i for all p_i.

We assume that all events are unique and all messages are distinguished. An event $([receive, m], p_k)$ is the *complement* of the event $([send, m, p_k], p_i)$ in a computation x iff both events are in x. An event $([send, m, p_k], p_i)$ is *fulfilled* in a computation x if it is in x and its complement event $([receive, m], p_i]$ is also in x. That is, the message m sent from process p_i to process p_k has already arrived. An event $([send, m, p_k], p_i)$ is *unfulfilled* in a computation x if it is in x, and it is not *fulfilled* in x.

Definition: x and y are *equivalent w.r.t.* p_i, denoted by $x \overset{i}{\sim} y$, iff $x_i = y_i$.

Note that the relation $\overset{i}{\sim}$ is an equivalence relation over system computations. For a computation x and p_i, we define the extensions of x which only have events on p_i.

Definition: $Extensions(x, i) \equiv \{y | y \text{ is an extension of } x \text{ and } x \overset{j}{\sim} y \text{ for all } j \neq i\}$.

Process p_i *reads* input a in x iff the input event $([input, a], p_i)$ is in x. Process p_i *decides* on d in x iff the decision event $([decide, d], p_i)$ is in x. A computation x is *i-input* iff for some value a, p_i reads input a in x. A computation x is *i-decided* iff for some value d, p_i decides on d in x. We assume that a process may read and decide only once.

A *protocol* $P = (C, N)$ *solves* a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ iff (1) For every input vector $\vec{a} \in \bar{I}$, and for every decision vector $\vec{d} \in T(\vec{a})$, there exists a computation $z \in C$ such that in z processes p_1,...,p_n read input values a_1,...,a_n and decide on d_1,...,d_n; (2) For every computation $z \in C$ such that in z processes p_1,...,p_n read input values a_1,...,a_n and decide on d_1,...,d_n, if $\vec{a} \in \bar{I}$ then $\vec{d} \in T(\vec{a})$; and (3) In any "sufficiently long" computation on input in $\bar{I}$ all processes decide (this last requirement is to be defined precisely later). It is also possible to define solvability so that (1) is replaced by the requirement that for each input vector $\vec{a} \in \bar{I}$, there exists a computation with $\vec{a}$ as input. In such a case we will say that a protocol P *minimally solves* a problem T.

We define when a set of input events is consistent. Intuitively, this is the case when all the input events in the set can happen in the same computation. Let P be a protocol that solves a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ For any input vector $\vec{a} \in \bar{I}$, the set $([input, a_1], p_1)$,..., $([input, a_n], p_n)$ is a consistent set of input events (w.r.t. T); and any subset of a consistent set of input events is also consistent. For simplicity of presentation, we assume that in any given computation the set of input events is consistent. An input event is said to be consistent with a *computation* y if it is consistent with the input events of y. Throughout the paper we consider a single protocol, $P = (C, N)$, that solves a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ and all references are made to that protocol.

3 Asynchronous Environment

In this section asynchronous message passing systems are characterized by stating six properties that any protocol which is operating in such an environment should satisfy.

The description considered here is similar to that of [Ta] and is based on [CM1,CM2]. We first introduce the notion of an *enabled* process. Process p_i is *enabled* at computation x iff there exists an event (e, p_i) such that $\langle x; (e, p_i)\rangle$ is a computation.

Definition: An *asynchronous protocol* is a protocol that satisfies the properties,

P1 Every prefix of a computation is a computation.

P2 Let x and y be computations such that y includes x and $x \overset{i}{\sim} y$. If $\langle x; (e, p_i)\rangle$ is a computation and (e, p_i) is not an input event which is inconsistent with y, then $\langle y; (e, p_i)\rangle$ is also a computation.

P3 For any computation x, process p_i and input value a, if the set of all input events in x together with $([input, a], p_i)$ is consistent, then there exists y in $Extensions(x, i)$, such that $([input, a], p_i)$ appears in y.

P4 For any computation $y = \langle x; (e, p_h)\rangle$ where (e, p_h) is an input event, any process $p_i \neq p_h$, and any computation $z \in Extensions(y, i)$, the sequence $\langle x; (z - y)\rangle$ is also a computation.

P5 For an *unfulfilled* event $([send, m, p_k], p_i)$ in a computation x, there exists an extension y of x such that $y \in Extensions(x, k)$, and $([send, m, p_k], p_i)$ is $fulfilled$ in y.

P6 For a computation x and an event $([receive, m], p_k)$, the sequence $\langle x; ([receive, m], p_k)\rangle$ is a computation only if $([receive, m], p_k)$ is the complement of some *unfulfilled* event in x.

Intuitively, property $P2$ means that if an event (e, p_i) can happen at a process p_i at some point in a computation, then the same event can happen at a later point, provided that p_i has taken no steps between the two points, and either it is not an input event or it is an input event consistent with the input events up to the later point. Property $P3$ means that a process which has not yet read an input value, may read any of the input values which do not conflict with the input values already read by other processes. Property $P4$ means that an input event on one process does not enable events of some other process. Property $P5$ means that it is always possible for a process to receive a message sent Property $P6$ means that a message is received only if it was sent previously and that it cannot be received twice.

Observation 1: In any asynchronous protocol, for computations x, y, z, if $y \in Extensions(x, i)$, z is an extension of x such that $x \overset{i}{\sim} z$ and y does not contain input events that are inconsistent with z, then $\langle z; (y - x)\rangle$ is a computation.

Observation 2: For any computations x and y, and for any process p_i; if y is an extension of x, p_i is *enabled* at x, and $x \overset{i}{\sim} y$ then p_i is *enabled* at y.

4 Decisions and Robustness

In this section we identify two classes of protocols, called *decision(t)* protocols, and *robust(t)* protocols, with t an integer, $0 \leq t < n$. In a decision(t) protocol every process tries to decide on a certain value, and additional conditions hold, to be defined below.

A decision is irreversible, that is, once a process decides on a value, the decision value cannot be changed. The important features of such protocols are the requirements on the possible input and decision (output) values. Here we require that there exists a group of at least $n-t$ processes and there exist input values such that after all the $n-t$ processes read these input values, the eventual decision value of at least one of them is still not uniquely determined. Compared with the usual requirement in other works where the above group should include all the processes, this requirement is very weak.

Typical examples of such protocols are those that solve any of the problems described in the Introduction and Section 2, where various assumptions, depending on the value of t, are made about the set of input vectors for each of these problems.

The following definition generalizes the notion of *valency* of a computation from [FLP]. Let d be a possible decision value and let U, W be sets of decision values.

Definition: A computation x is (i, W)-*valent* iff (1) for every $d \in W$, there is an extension z of x such that p_i decides on d in z, and (2) for every $d \notin W$, there is no extension z of x such that p_i decides on d in z.

A computation is *i-univalent* iff it is (i,d)-*valent* for some (single) value d. It is *i-multivalent* otherwise. It is *univalent* iff it is *i-univalent* for all $i = 1..n$. As we shall see later no computation in a protocol studied here is $(i, \emptyset)$-*valent*.

Lemma 1: In any asynchronous protocol, if x is *(i,W)-valent* computation, y is *(i,U)-valent* computation, and y *includes* x then $U \subseteq W$.

Proof: We prove that for an arbitrary value d, $d \in U$ implies $d \in W$. Assume $d \in U$. By the definition of *valency* there exists an extension z of y such that p_i decides on d in z. Clearly, the computation z includes x. It follows from properties $P1$, $P2$ and $P3$ that there exists an extension z' of x such that $z \overset{i}{\sim} z'$. (To see this, take z' as the concatenation of x with the subsequence of all events in z which are not in x.) Since, p_i decides on d in z and $z \overset{i}{\sim} z'$ it follows (from the definition of *decides*) that p_i decides on d in z'. Since z' is an extension of x, $d \in W$. □

Definition: Let y and y' be *(i,W)-valent* and *(i,W')-valent*, respectively. Then y and y' are *i-compatible* iff $W \cap W' \neq \emptyset$. They are *compatible* iff they are *i-compatible* for all $i = 1..n$.

Using the above notions we are now able to characterize decision(t) protocols formally.

Definition: A *decision(t) protocol* is a protocol that satisfies the requirement:

N(t): There exists a computation x, process p_j and set of processes Q, such that $|Q| \geq n - t$, for every $p_i \in Q$ x is *i-input*, $p_j \in Q$, and x is *j-multivalent*. (non-triviality.)

It is not difficult to see why any protocol that solves the variant of the consensus problem which is mentioned in the introduction (i.e., with the promise that for each input vector, $|\#1 - \#0| \geq t$) must satisfy $N(t)$. As will be shown later in Section 7, any problem that can be solved by a protocol that does not satisfy $N(t)$, can also be solved by a trivial protocol in which every process decides after having $n-t$ input values. Notice that $N(t)$ implies $N(t+1)$. This requirement generalizes a requirement from Lemma 2 of [FLP], that a nontrivial consensus protocol must have a bivalent initial configuration.

Next we identify the class of *robust(t)* protocols. A failure of a process means that no subsequent event can happen on this process. This is a very weak type of failure, called

crash failure. Since we want to prove an impossibility result, it follows that if the result holds for crash failures it also holds for any stronger type of failure.

In order to define robust(t) protocols formally, we need the concept of a *Q-fair sequence*. Let Q be a set of processes, a *Q-fair sequence* w.r.t. a given protocol is a (possibly infinite) sequence of events, where: (1) Each finite prefix is a computation; (2) For an *enabled* process $p_i \in Q$ at some prefix x, there exists another prefix y which is an extension of x such that there is an event (e, p_i) in $(y-x)$; (3) For any event $([send, m, p_k], p_i)$ which is *unfulfilled* in some prefix, if process $p_k \in Q$, then $([send, m, p_k], p_i)$ is *fulfilled* in another prefix; (4) The sequence $\langle x; ([receive, m], p_k)\rangle$, is a prefix only if $([receive, m], p_k)$ is the complement of some *unfulfilled* event in x, and (5) Every process p_k not in Q appears only finitely often in the sequence.

A Q-fair sequence captures the intuition of an execution where all enabled processes which belong to Q can proceed, all messages sent to processes belonging to Q are eventually delivered and a message is received only if it was sent previously. By (5), a process not in Q eventually either is not enabled or simply stops taking actions. Notice that a Q-fair sequence may be *infinite* and in such a case it is not a computation. It follows from Observation 2, $P5$ and $P6$ that for every set of processes Q, any computation is a prefix of a Q-fair sequence. Requirement (4) follows from $P6$ and requirement (1).

Definition: A *robust(t) protocol* is a protocol that satisfies the requirement:

R(t): For every set Q of processes where $|Q| \geq n - t$, every *Q -fair sequence* has a finite prefix that is *i -decided* for every $p_i \in Q$.

Note that the class of robust(t+1) protocols is included in the class of robust(t) protocols. Furthermore, from examples of protocols which are robust(t) but not robust($t+1$), we can see that the inclusion is strict. Requirement $R(0)$ formally expresses requirement (3) from the definition of *solves* given in Section 2. It follows from $R(0)$ and from the fact that every computation is a prefix of some N-fair sequence that (in asynchronous robust(0) protocols) no computation is $(i, \emptyset)$-*valent*.

Lemma 2: In any asynchronous robust(0) protocol, for any *i-input* computation x, for any computation $y \in Extensions(x, i)$ and for any extension z of x, if $x \overset{i}{\sim} z$ then y and z are *compatible*.

Proof: Assume $x \overset{i}{\sim} z$. From Observation 1, $\langle z; (y - x)\rangle$ is a computation. The computation $\langle z; (y - x)\rangle$ includes both y and z. Thus, from Lemma 1 and the fact that no computation is $(i, \emptyset)$-*valent* (for all $i = 1..n$), y and z are *compatible*. □

5 Splitters and Deciders

In this section we prove that for any asynchronous, non-trivial protocol there is a computation in which some process is a *splitter*. This process can split the possible outputs of the protocol to two disjoint sets. This result holds both for fault-free and for fault-tolerant protocols. In fault-tolerant protocols, this splitter must be a *decider*, that can split its own output values into two different singletons. Our Decider Theorem generalizes a previous result by Chandy and Misra [CM1, Theorem 1], in which they proved a similar condition for commit protocols in the presence of a single faulty process. A result resembling our Decider Theorem was presented independently in [BW].

Let us consider the eight requirements mentioned so far. All of them capture very natural concepts: $P1 - P6$ and $R(t)$ express the well known notions of asynchronous and robust protocols respectively, while $N(t)$ requires that a given solution is not trivial. This motivates the question of what can be said about protocols that satisfy all the above requirements. For later reference we call these protocols Decision(t) Asynchronous Robust(t) Protocols (abbv. DEAR(t) P's).

In order to state our result, we first define the notions of a *j-splitter* and a *decider*. Informally, a process p_i is a *j-splitter* at a computation x, if it is possible to extend x with two sequences of events on process p_i only, in such a way that the (single) value some process p_j can (still) decide on in each of the resulting two computations, differ. In particular when process p_i is an *i-splitter*, we say that p_i is a *decider* at x.

Definition: A process p_i is a *j-splitter* at a computation x iff there exist two *j-univalent* computations y and y', both belonging to $Extensions(x, i)$, such that y and y' are not *j-compatible*. A process p_i is a *decider* at a computation x iff p_i is an *i-splitter* at a computation x.

Lemma 3 (The Splitter Lemma): In any asynchronous robust(0) protocol, if x is a *j-multivalent* computation then there is an extension v of x and a process p_i such that p_i is a *j-splitter* at v.

Proof: Let x be a *j-multivalent* computation. We must show an extension v of x and a process p_i such that $Extensions(v, i)$ contains *j-univalent* computations y and y' which are not *j-compatible*. We use the following definition. Computation x is *j-open w.r.t. process* p_i iff any $y \in Extensions(x, i)$ is *j-multivalent*. We consider two cases.

Case 1: Assume that there exists a *j-multivalent* extension x' of x and a process p_i such that any extension v of x' is not *j-open w.r.t. process* p_i.

Let v be an extension of x' as above. We denoted by $\Phi(v)$, one (arbitrary) *j-univalent* extension of v, with events only on p_i (such an extension must exist because v is not *j-open w.r.t.* p_i). If v is not *i-input*, we choose an extension $\Phi(v)$ that includes an input event on p_i (this is possible due to $P3$). Since x' is *j-multivalent*, there is an extension z of x' ($z \neq x'$) such that z and $\Phi(x')$ are not *j-compatible*. Consider the extensions of x' which are also prefixes of z. Since z and $\Phi(x')$ are not *j-compatible*, there exist two extensions y and y' of x' such that $\Phi(y)$ and $\Phi(y')$ are not *j-compatible*, and where y is a one event extension of y'. Therefore $y = \langle y'; (e, p_h)\rangle$ for some event (e, p_h).

If $p_i = p_h$, then $\Phi(y)$ and $\Phi(y')$ both belong to $Extensions(y', i)$, and the lemma is proven. If $p_i \neq p_h$, and either y' is *i-input* or (e, p_h) is not an input event, then by Observation 1, $w = \langle y; (\Phi(y') - y')\rangle$ is a computation. From the construction, $w \in Extensions(y, i)$. Since w includes $\Phi(y')$ (which is *j-univalent*), by Lemma 1 w is also *j-univalent*, with the same value. Moreover, since $\Phi(y)$ and $\Phi(y')$ are not *j-compatible*, it follows that $\Phi(y)$ and w are also not *j-compatible*, and the Lemma again holds.

If $p_i \neq p_h$, and both (e, p_h) is an input event on p_h, and y' is not *i-input*, then the w above may not be a computation, because the input event (e, p_h) could be inconsistent with an input event on p_i. However, by $P4$, $w' = \langle y'; (\Phi(y) - y)\rangle$ (i.e., not including the input event on p_h) is a computation. By $P2$ and the definition of Φ, $\langle \Phi(w'); (e, p_h)\rangle$ (i.e., adding the input event on p_h) is a *j-univalent* computation (here we use the fact that $(\Phi(y) - y)$ has an input event on p_i which *is* consistent with (e, p_h)). By Lemma 1, $\langle \Phi(w'), (e, p_h)\rangle$ is *j-compatible* with $\Phi(w')$. Since $\langle \Phi(w'); (e, p_h)\rangle$ includes $\Phi(y)$, also $\Phi(w')$

and $\Phi(y)$ are *j-compatible*. This, and the fact that $\Phi(y)$ and $\Phi(y')$ are not *j-compatible*, imply that $\Phi(w')$ and $\Phi(y')$, which both belong to $Extension(y', i)$, are both *j-univalent* and are not *j-compatible*. This implies the lemma.

Case 2: The negation of Case 1. That is, assume that for every *j-multivalent* extension x' of x and for every p_i, there exists an extension v of x' which is *j-open w.r.t.* p_i.

Using this assumption, we derive a contradiction similar to that in [FLP] by constructing an N-fair sequence $\mathcal{F}$ such that all its finite prefixes are *j-multivalent*.

The construction of $\mathcal{F}$ is done in steps. At each step m we extend the *j-multivalent* sequence $\mathcal{F}_{m-1}$ constructed at step $m-1$ to a *j-multivalent* sequence $\mathcal{F}_m$. $\mathcal{F}_0$ is the given *j-multivalent* computation. At each step $m \geq 1$, we set $i = m \ (mod\ n)$. If p_i is not enabled at $\mathcal{F}_m$ then we let $\mathcal{F}_m = \mathcal{F}_{m-1}$. If p_i is enabled, we first extend $\mathcal{F}_{m-1}$ to a computation v which is *j-open w.r.t.* p_i. Since we can extend the computation v by an arbitrary (possible) event on p_i and still get a *j-multivalent* computation, we choose one such extension so that p_i eventually reads its input and receives each message sent to it. The resulting sequence is $\mathcal{F}_m$. Thus we can construct an N-fair sequence which is everywhere *j-multivalent*, contradicting $R(0)$. This case is therefore impossible. □

Lemma 4: In any asynchronous robust(1) protocol, for any *i-input* computation x and any process p_j where $i \neq j$, process p_i is not a *j-splitter* at x.

Proof: Let y and y' both belong to $Extensions(x, i)$. Apply $P1 - P6$ and $R(1)$ to conclude that there is a *j-univalent* extension z of x such that $x \overset{i}{\sim} z$. From Lemma 2, both y, z and y', z are compatible. Since z is *j-univalent*, y and y' are *j-compatible*. □

THEOREM 1 (The Decider Theorem): In any DEAR(t) P, there exists a *j-input* computation v and a process p_j, such that p_j is a *decider* at v.

Proof: Let x be a *j-multivalent* computation satisfying $N(t)$, and let Q be the corresponding set of processes, where $p_j \in Q$. By Lemma 3, there is an extension v of x and a process p_i such that p_i is a *j-splitter* at v. We must show that $i = j$. Assume, for contradiction, that $i \neq j$, and let y and y' be any two computations in $Extension(v, i)$. We will show that y and y' must be *j-compatible* if $j \neq i$, which will contradict the assumption that p_i is a *j-splitter* at v.

If the computation v above is *i-input*, then we are done by Lemma 4. Otherwise, v is not *i-input*, and by $N(t)$ $p_i \notin Q$, by $N(t)$. By $R(t)$, there is a *j-univalent* extension z of v such that only processes from Q are activated in $(z - v)$; in particular, $(z - v) \overset{i}{\sim} v$ and $(z - v)$ contains no input events (since those from Q were already in v).

Let $w = \langle y; (z - v) \rangle$ and $w' = \langle y'; (z - v) \rangle$. By Observation 1, both w and w' are computations. Also, since both w and w' include z, and since z is *j-univalent*, w and w' are *j-compatible*. Since w includes y and w' includes y', y and y' are also *j-compatible*. This is the desired contradiction, and the Theorem is proven. □

6 Robust(t) Asynchronous Dependent(t) Protocols

In the previous sections we have defined several classes of protocols. In this section we investigate the class of robust(t) asynchronous dependent(t) protocols (abbv. ROAD(t) P's). The class of ROAD(t) P's is defined by the previous eight axioms, plus the following requirement, which we call *dependency*.

Definition: A *dependent(t) protocol* is a protocol that satisfies the requirement:

D: For any computation x, if every process has read an input value in x and x is a j-*decided* computation for all $j = 1..i-1, i+1..n$ then x is i-*univalent*. (dependency.)

Requirement D means that as soon as all processes except one have made their decisions, and all processes have read their input, the eventual decision value of the remaining process is uniquely determined. All protocols which solve the problems mentioned in the Introduction and in Section 2 satisfy D. Notice that, for two *univalent* computations x and y which are *j-compatible* for all $j = 1..i-1, i+1..n$, it does not follow from requirement D that x and y are also *i-compatible*.

We prove in this section that for any $1 \leq t \leq n$, the class of ROAD(t) P's is an *empty* class. Put another way, we show that there does not exist any ROAD(t) P.

Lemma 5: In any ROAD(t) P, for any *i-input* computation x, there exists a *univalent* extension z of x such that $x \overset{i}{\sim} z$.

Proof: It follows from Observation 2, $P5$ and $P6$, that for any process p_i, any computation x is a prefix of some $(N-\{p_i\})$-fair sequence. Apply requirement $R(1)$ to the above sequence to conclude that, for any computation x and any process p_i, there exists an extension z' of x such that $x \overset{i}{\sim} z'$ and z' is *j-decided* for any $j \neq i$. By $P3$ there is an extension z of z' in which all processes read their input and $x \overset{i}{\sim} z$. From D the computation z is *univalent*. □

Lemma 6: In any ROAD(t) P, for any *i-input* computation x, any two computations y and y' which belong to $Extensions(x, i)$ are *compatible*.

Proof: Let y and y' both belong to $Extensions(x, i)$. Apply Lemma 5 to conclude that there is a *univalent* extension z of x such that $x \overset{i}{\sim} z$. From Lemma 2, both y, z and y', z are compatible. Since z is *univalent*, y and y' are *compatible*. □
Now the main theorem can be stated and proven.

THEOREM 2: There is no ROAD(t) P.

Proof: By Theorem 1 there exists a computation x and process p_i for which p_i is a decider at x. However, this means precisely that there are two computations in $Extensions(x, i)$ which are not compatible, contradicting Lemma 6. □

7 Dependent(t) Problems

In this section we identify the problems that cannot be solved in an unreliable asynchronous message passing system. We do this by identifying those problems which are solved only by ROAD(t) protocols. Hence, the impossibility of solving these problems will follow from Theorem 2.

We say that a problem can be solved in an environment where t processes may fail, if there exists a robust(t) protocol that solves it. Since we assume that up to t processes may fail, any protocol that solves a problem should satisfy properties $P1 - P6$, and the requirement $R(t)$. Thus, we are now left with the obligation of identifying those problems which force any protocol that solves them also to satisfy requirements $N(t)$ and D. Let Q

denote a set of processes, and $\vec{v}$ and $\vec{v'}$ be vectors. We say that $\vec{v}$ and $\vec{v'}$ are *Q-equivalent*, if they agree on all the values which correspond to the indices (of the processes) in Q. A set of vectors H is *Q-equivalent* if any two vectors which belong to H are *Q-equivalent*. Also, we define: $T(H) \equiv \bigcup_{\vec{a} \in H} T(\vec{a})$.

Definition: A problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ is a *dependent(t) problem* iff it satisfies:

T1(t): There exists a set of processes Q where $|Q| \geq n-t$, and there exists a *Q-equivalent* set $H \subseteq \bar{I}$ such that $T(H)$ is not a *Q-equivalent* set.

T2: For every $\vec{a} \in \bar{I}$, every set of processes Q where $|Q| = n-1$, and every two different decision vectors $\vec{d}$ and $\vec{d'}$, if both $\vec{d}$ and $\vec{d'}$ belong to $T(\vec{a})$ then they are not *Q-equivalent*.

Requirement $T1(t)$ means that $n-t$ input values (in an input vector) do not determine the corresponding $n-t$ decision values (in the decision vectors). Any problem that does not satisfy requirement $T1(t)$ can easily be solved in a completely asynchronous environment where t processes may fail. (Each process sends its input value to all other processes, then it waits until it receives $n-t$ values; assuming it does not satisfies $T1(t)$, it has now enough information to decide.) Note that $T1(t)$ implies $T1(t+1)$. Requirement $T2$ means that a single input vector cannot be mapped into two decision vectors that differ only by a single value.

THEOREM 3: A dependent(t) problem cannot be solved in an asynchronous message passing system where up to t failures may occur.

A problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ *includes* a problem $T : \bar{I'} \rightarrow 2^{\bar{D'}} - \{\emptyset\}$ iff $\bar{I'} = \bar{I}$, and for every $\vec{a'} \in \bar{I'}$: $T'(\vec{a'}) \subseteq T(\vec{a'})$. It is easy to see that a protocol P *minimally solves* a problem T iff there exists a problem T' which is included in T such that P *solves* T'. A problem $T' : \bar{I} \rightarrow 2^{\bar{D'}} - \{\emptyset\}$ is a *sub-problem* of a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ iff $\bar{I'} \subseteq \bar{I}$, and for every $\vec{a'} \in \bar{I'}$: $T(\vec{a'}) = T'(\vec{a'})$. It is easy to see that if a protocol P *solves* (*minimally solves*) a problem T then P *solves* (*minimally solves*) any sub-problem T' of T.

8 Discussion

We showed that for the message passing model there exists a resiliency hierarchy. That is, for each $0 \leq t < n-1$ there are problems that can be solved in the presence of $t-1$ failures but can not be solved in the presence of t failures.

The identities of processes have been used by us for the purposes of the metadiscussion in the proofs. However, no assumption has been made about whether or not process ids are known (available) to the processes themselves. Thus the impossibility result we have proven holds even for the strongest possible assumption about identifiers, namely that all processes have distinct identifiers which are universally known.

It is simple to modify the presentation to allow an atomic *broadcast* instead of the usual *send* event. Also, by simply modifying property $P6$, we can show that the result holds even under the assumption that messages sent from one process to another are received in the order they were sent (i.e., FIFO). Few more generalizations of the results mentioned so far are included in [TKM1, Section 8].

Acknowledgements: We are most grateful to Nissim Francez, Oded Goldreich, Joe Halpern, and Yaron Wolfstahl for many helpful discussions concerning this work.

References

[ABDKPR] Attiya, H., Bar-Noy, A., Dolev, D., Koller, D., Peleg, D., and Reischuk, R. Achievable cases in an asynchronous environment, *ACM-FOCS* 1987, 337-346.

[BMZ] Biran, O., Moran S., and Zaks, S. A Combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor, *ACM-PODC* 1988, 263-275.

[BW] Bridgland, M., and Watro, R. Fault-tolerant decision making in totally asynchronous distributed systems, *ACM-PODC* 1987, 52-63.

[CM1] Chandy, M., and Misra, J. On the nonexistence of robust commit protocols, Unpublished manuscript, November 1985.

[CM2] Chandy, M., and Misra, J. How processes learn, *Distributed Computing* 1986, 40-52.

[DDS] Dolev, D., Dwork, C., Stockmeyer, L. On the minimal synchronism needed for distributed consensus, *JACM 34*, 1, 1987, 77-97.

[DLS] Dwork, C., Lynch, N., Stockmeyer, L. Consensus in the presence of partial synchrony, *JACM 35*, 2, 1988, 288-323.

[ESY] Even, S., Selman A., and Yacobi, Y. The complexity of promise problems with applications to public-key cryptography, *Information and Control* 1984, 159-173.

[FLP] Fischer, M., Lynch, N., Paterson, M. Impossibility of distributed consensus with one faulty process, *JACM 32*, 2, 1985, 374-382.

[MW] Moran, S., and Wolfstahl, Y. An extended impossibility result for asynchronous complete networks, *IPL 26*, November 1987, 145-151.

[Ta] Taubenfeld, G. Impossibility Results for Decision Protocols, Technion Technical Report #445, January 1987. Revised version, Technion TR #506, April 1988.

[TKM1] Taubenfeld, G., Katz, S., and Moran, S. Impossibility results in the presence of multiple faulty processes, Technion Technical Report #492, January 1988.

[TKM2] Taubenfeld, G., Katz, S., and Moran, S. Initial failures in distributed computations, Technion Technical Report #517, August 1988.

[TM] Taubenfeld, G., and Moran, S. Possibility and impossibility results in a shared memory environment, To appear in *3rd International workshop on distributed algorithms*, Nice, France, September 1989. Also, *Yale technical report YALEU/DCS/TR–708* (May 1989).