IMPOSSIBILITY RESULTS IN THE PRESENCE OF MULTIPLE FAULTY PROCESSES

by

G. Taubenfeld, S. Katz and S. Moran

Technical Report #492

January 1988
Impossibility Results in the Presence of Multiple Faulty Processes

by

Gadi Taubenfeld, Shmuel Katz and Shlomo Moran

Computer Science Department
Technion, Haifa 32000
Israel

Abstract

We investigate the impossibility of solving certain problems in an unreliable distributed system where multiple processes may fail. A necessary condition is provided for the solvability of problems in the presence of multiple faulty processes, and variants of problems which are unsolvable in the presence of a single faulty process (such as consensus, choosing a leader, ranking, matching) are shown to be solvable in the presence of \( t-1 \) faulty processes but not in the presence of \( t \) faulty processes for any \( t \). In order to prove the impossibility result we use a novel technique: a contradiction is shown among a set of axioms which characterize any fault-tolerant protocol solving the problems we treat. These results generalize previously known impossibility results for completely 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. Let \( n \) denote the total number of processes. For each natural number \( t \) such that \( 1 \leq t < n \), we will 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.

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 (i.e. a protocol which has all processes decide on the same value) that tolerates even a single process failure. This fundamental result has been extended to other models of computation [DDS,DLS]. The relation of our work to some results in [DDS] is discussed in the last section. 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].

Define an input vector to be a vector \( \bar{a}=(a_1, \ldots, a_n) \), where \( a_i \) is the input value of process \( p_i \). A crucial assumption in all the above impossibility 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. (Each process outputs its input value.) One of the consequences of our result is to identify a property (or a promise [ESY]) which a set of input vectors (of a problem) should satisfy so that the problem cannot be solved in the presence of \( t \) failures.

We show variants of the problems which are known to be unsolvable (in a nontrivial way) 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 can not 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 Section 8, this second requirement is weakened and incorporated into a single requirement which extends the basic result.

In order to prove the above result for protocols, we use an axiomatic approach for proving properties of protocols (and problems) which is 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 five 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 eight 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 dependency and robustness are introduced. In Section 5, a special class of protocols is identified and it is proved that all its members cannot tolerate \( t \) process failures. In Section 6, we present the result of the previous section from another point of view and show that in any non-trivial fault-
tolerant protocol there must exist a process (which is called a decider) that has some special property. In Section 7, we finally identify the class of problems that can not be solved in the presence of t process failures. In Section 8, we extend the results presented in previous sections. The relation among the classes of protocols and problems we define are summarized in Section 9.

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 \( \overline{I} \) and \( \overline{D} \) be subsets of \( I^n \) and \( D^n \), respectively. A problem \( T \) is a mapping \( T: \overline{I} \rightarrow 2^{D^*} \) which maps each \( n \)-tuple in \( \overline{I} \) to subsets of \( n \)-tuples in \( \overline{D} \). We call the vectors \( \vec{a} = (a_1, \ldots, a_n) \) where \( \vec{a} \in \overline{I} \), and \( \vec{d} = (d_1, \ldots, d_n) \) where \( \vec{d} \in \overline{D} \), 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, which we will also refer 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 \( P_i \) (\( i = 1 \ldots n \)) decides on a value \( v_i \) from \( D \), \( D = \{ 1, \ldots, n \} \), 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 (abbrv. processes), \( N = \{ P_1, \ldots, P_n \} \). A computation is a finite sequence of events. There are four types of events: send, receive, input, and decide. A send event, denoted \( ([\text{send}, m, p_k], P_i) \), represents sending a message \( m \) to process \( P_k \) by process \( P_i \). A receive event, denoted \( ([\text{receive}, m], p_k) \), represents receiving a message \( m \) by process \( P_k \). An input event, denoted \( ([\text{input}, a], P_i) \), represents reading an input value \( a \) by process \( P_i \). A decide event, denoted \( ([\text{decide}, d], P_i) \), represents deciding on a decision value \( d \) by process \( P_i \). One may also consider an internal event in which a process executes some local computation; however nowhere else in this paper do we need to refer to such an event. 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 \( Q \subseteq \mathbb{N} \), and \( |Q| \) denotes the cardinality of \( Q \). The symbols \( x,y,z \) denote computations. Also \( <x;y> \) 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 \( p_i \) as the process component (i.e., \( x_i \) is the sequence of all events in \( x \) which are on process \( p_i \)).

Definition: \( y \) includes \( x \) iff \( x_i \) is a prefix of \( y_i \) for all \( p_i \) (the relation \( \text{includes} \) is a generalization of \( \text{extension} \)).

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

Definition: Computations \( x \) and \( y \) are equivalent w.r.t. \( p_i \), denoted by \( x \equiv y \) iff \( x_i = y_i \).

Note that the relation \( \equiv \) is an equivalence relation over system computations. Also, for \( x \) a prefix of \( y \), there is an event on \( p_i \) between \( x,y \) iff \( -(x \equiv y) \). For a computation \( x \) and process \( p_i \), we define the extensions of \( x \) which only have events on \( p_i \).

Definition: Extensions \( (x,i) \equiv \{ y \mid y \text{ is an extension of } x \text{ and } x \not\equiv y \text{ for all } j=1..i-1,i+1..n \} \).

Process \( p_i \) reads input \( a \) in a computation \( x \) iff the input event \( ([\text{input},a],p_i) \) is in \( x \). Process \( p_i \) decides on \( d \) in a computation \( x \) iff the decision event \( ([\text{decide},d],p_i) \) is in \( x \). A computation \( x \) is \( i\text{-input} \) iff for some value \( a,p_i \) reads input \( a \) in \( x \). A computation \( x \) is an input computation iff it is \( i\text{-input} \) for all \( i=1..n \). A computation \( x \) is \( i\text{-decided} \) iff for some value \( d,p_i \) decides on \( d \) in \( x \). A computation \( x \) is decided iff it is \( i\text{-decided} \) for all \( i=1..n \).

Since we are interested in protocols which solve the type of problems mentioned previously, without loss of generality, we can assume that a process may read and decide only once. More formally, for any computation \( x \) and for any process \( p_i \) there is at most one input event, and at most one decision
event on process $p_i$ in computation $x$. Three simple observations which follow from the definitions given so far are: (1) For computations $x$ and $y$, if $y_i$ is an extension of $x_i$ and $p_i$ reads input $a$ in $x$ then $p_i$ reads input $a$ in $y$; (2) For computations $x$ and $y$, if $y_i$ is an extension of $x_i$ and $p_i$ decides on $d$ in $x$ then $p_i$ decides on $d$ in $y$; (3) For a computation $<x; (e, p_i)>$, if $x$ is an $i$-input computation then $(e, p_i)$ is not an input event.

A protocol $P = (C, N)$ solves a problem $T: I \rightarrow 2^D - \{\emptyset\}$ iff (1) For every input vector $\vec{a} \in I$, 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 $\vec{a} \in I$ then $\vec{d} \in T(\vec{a})$; and (3) In any "sufficiently long" computation on input in $I$ 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 $\vec{a} \in I$, there exists a computation with $\vec{a}$ as input. In such a case we will say that a protocol $P$ minimally solves a problem $T$. The difference between the two is that in the former case every possible decision vector is the result of some computation, while in the later this is not so. It will be shown in section 7 that it is sufficient to prove the impossibility result for the former definition of solvability, and then to derive from it a result for the later case.

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 $T: I \rightarrow 2^D - \{\emptyset\}$ For any input vector $\vec{a} \in I$, 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. Throughout the paper we consider a single protocol, $P = (C, N)$, that solves a problem $T: I \rightarrow 2^D - \{\emptyset\}$ and all references are made to that protocol.

3. Asynchronous Environment

In this section asynchronous message passing systems are characterized by stating five properties that any protocol which is operating in such an environment should satisfy. The formal description of asynchronous message passing systems considered here is based on [CM1,CM2]. We first introduce the notion of an active process. Process $p_i$ is active at computation $x$ if there exists an event $(e, p_i)$ such that $<x; (e, p_i)>$ is a computation.
Definition: An asynchronous protocol is a protocol that satisfies the following properties,

P1: Every prefix of a computation is a computation.

P2: Let \(<x; (e, p_i)>) be a computation where \((e, p_i)\) is not an input event, and let \(y\) be a computation which includes \(x\) such that \(x \subseteq y\); then, \(<y; (e, p_i)>\) is a computation.

P3: For any computation \(x\), any process \(p_i\) and any input value \(a\); if the set of all input events in \(x\) together with the input event \(((input, a), p_i)\) is consistent then \(<x; ((input, a), p_i)>) is a computation.

P4: 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 \(y\).

P5: For a computation \(x\) and an event \(((receive, m), P_k)\). The sequence \(<x; ((receive, m), P_k)>) is a computation only if \(((receive, m), P_k)) is the complement of some unfulfilled event in \(x\).

We note that from property P1 the empty sequence (denoted by null) is a computation. 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 it is not an input event, and \(p_i\) has taken no steps between the two points. 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. For example, if we assume that the input values different processes may read in the same computation are distinct, then a process may read any value which has not already been read by other processes. Notice that if \(x\) is i-input then no input event on \(p_i\) is consistent with the set of input events in \(x\). Property P4 means that it is always possible for a process to receive a message sent to it. Property P5 means that a message is received only if it was sent previously and that it cannot be received twice.

The following observation follows from properties P1, P2 and the observations mentioned in the previous section.

Observation 1: In any asynchronous protocol, for computations \(x, y, z\), where \(x\) is i-input, if \(y \in Extensions(x, i)\) and \(z\) is an extension of \(x\) such that \(x \subseteq z\) then \(<z; (y-x)>) is a computation.

The next observation follows from properties P2 and P3. It says that a process cannot become passive (i.e., not active) as a result of an event on some other process. In particular, if a process is "ready to read" an input value then an input event on some other process cannot prevent this process from reading some input value (although it may prevent this process from reading a specific input value which it could read
Observation 2: For any computations $x$ and $y$, and for any process $p_i$; if $y$ is an extension of $x$, $p_i$ is active at $x$, and $x \subseteq y$ then $p_i$ is active at $y$.

4. Dependency and Robustness

In this section we identify two classes of protocols, called dependent($t$) protocols, and robust($t$) protocols. In a dependent($t$) 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. For the input, it is required 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 (i.e., be of size $n$), this requirement is very weak. The requirement for the decision values is that the eventual decision value of any (single) process is uniquely determined when all other processes have already decided.

Typical examples of dependent($t$) protocols are the protocols that solve any of the problems described in Section 2, where various assumptions, depending on the value of $t$, are made about the set of input vectors for each of these problems. Having that class formally defined, we will prove in the next section that for each natural number $t$, where $1 \leq t \leq n$ ($t \neq 0$), no protocol in the class of dependent($t$) protocols can tolerate $t$ process failures.

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 $d \not\in W$, 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 $i=1..n$. It is multivalent otherwise. There is a major difference between the notions of $i$-decided and $i$-univalent. It is impossible that a process will
become \( i-decided \) (at a computation) as a consequence of an action of some other process. However, a process may become \( i-univalent \) (i.e., its ultimate decision value can be uniquely determined) as a consequence of some other process' action. That is, it is possible to have two computations \( x \) and \( y \) such that \( x \perp y \), \( x \) is \( i-univalent \) and \( y \) is \( i-multipotent \). Also, for any computation \( x \) and any process \( p_i \), if \( x \) is \( i-decided \) then \( x \) is \( i-univalent \) (but not vice versa). Note that for any computation \( x \) and process \( p_i \) there exists a single set \( W \) such that \( x \) is \((i,W)-valent\).

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 \( P_1, P_2 \) and \( P_3 \) that there exists an extension \( z' \) of \( x \) such that \( z \perp z' \). (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 \( z \perp z' \) it follows (from the definition of decides) that \( p_i \) decides on \( d \) in \( z' \). Since \( z' \) is an extension of \( x \), \( d \in W \). \( \square \)

Intuitively, the above lemma merely states that the set of possible decision values does not increase as the execution proceeds.

Definition: A dependent \((t)\) protocol is a protocol that satisfies the requirements:

1. \( D_1(t) \): There exists a computation \( x \), process \( p_i \) and set of processes \( Q \), such that \( |Q| \geq n-t \), for every \( p_j \in Q \) \( x \) is \( j-input \), \( p_i \in Q \), and \( x \) is \( i-multivalent \).
D2: For any input computation \( x \), if \( x \) is a \( j \)-decided computation for all \( j = 1..i-1,i+1..n \) then \( x \) is \( i \)-univalent. (dependency.)

Requirement \( D1(t) \) means that it is possible that \( n-t \) processes will read their input values and still, after that, the final decision value of at least one of these processes is not yet determined. For example, for any \( t \), \( D1(t) \) is clearly not necessarily satisfied by a protocol where each process is to decide on its own input. On the other hand, for any protocol solving the rotation problem (i.e., which decides on its neighbor's input), \( D1(0) \) does not hold, but \( D1(1) \) does. Also, 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, \( 1 \# 1-\#0 \geq t \)) must satisfy \( D1(t) \). Notice that \( D1(t) \) implies \( D1(t+1) \).

Note: It may be useful to explain requirement \( D1(t) \) using the notion of knowledge as defined by Chandy and Misra [CM2] (see also [HK,KT]). That is, for a set of processes \( Q \), predicate \( b \) and computation \( x \), \( [Q \text{ knows } b] \) at \( x \) iff for all \( y \) such that for any \( p_i \in Q \), \( x_i=y_i \) : \( b \) holds at \( y \). Requirement \( D1(t) \) means that there exists a computation \( x \), and a set of processes \( Q \) where \( |Q| \geq n-t \) such that all the processes read their input values (in \( x \)) and still for some \( p_i \in Q \), \( \neg [Q \text{ knows } i \text{-univalent}] \) at \( x \).

Requirement \( D2 \) 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 \( D2 \). 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 \( D2 \) that \( x \) and \( y \) are also \( i \)-compatible.

Next we identify the class of \( \text{robust}(t) \) protocols. In this paper we consider the possibility of \( t \) process failures (\( 0 \leq t \leq n \)). 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. Informally, a protocol can tolerate \( t \) faulty processes if in spite of a failure of any group of \( t \) processes at any point in the computation, each of the remaining processes eventually decides on some value. A protocol is \( \text{robust}(t) \) if it can tolerate \( t \) process failures.

In order to define \( \text{robust}(t) \) 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 active 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; and (4) The sequence \(<x;[(receive, m), p_k]>\), is a prefix only if \([(receive, m), p_k]\) is the complement of some unfulfilled event in \( x \).

A \( Q \)-fair sequence captures the intuition of an execution where all active 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. Notice that a \( Q \)-fair sequence may be infinite and in such a case it is not a computation. It follows from Observation 2, \( P4 \) and \( P5 \) that, in asynchronous protocols, for every set of processes \( Q \), any computation is a prefix of a \( Q \)-fair sequence. The requirement that all messages are received after they are sent, (requirement (4) in the definition of \( Q \)-fair sequence) follows from \( P5 \) and requirement (1).

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

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

Note that for any values \( t \) such that \( 0 \leq t < n \), requirement \( R(t+1) \) implies \( R(t) \), i.e., 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 \( R(0) \) means that in any "long enough" execution of a protocol, if no process fails then each process (eventually) decides on a value. In fact, \( R(0) \) formally expresses requirement (3) from the definition of solves given in Section 2. Thus, any protocol that solves a problem should (by definition) satisfy \( R(0) \). 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 \((i, \mathcal{O})\)-valent.

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

\textbf{Proof:} Assume \( x \perp z \). From Observation 1, \(<z;(y-x)>\) is a computation. The computation \(<z;(y-x)>\) includes both \( y \) and \( z \). Thus, from Lemma 1 and the fact that no computation is \((i, \mathcal{O})\)-valent (for all \( i=1..n \)), \( y \) and \( z \) are compatible. \( \Box \)
5. 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 (abbr. ROAD(t) P's). The class of ROAD(t) P's is defined by the entire eight axioms. We assume that the symbol t stands for a natural number such that 1 \leq t \leq n (t \neq 0). We prove in this section that 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 3: In any ROAD(t) P, for any i-input computation x, there exists a univalent extension z of x such that x \perp z.

Proof: It follows from Observation 2, P 4 and P 5, that for any process pi, any computation x is a prefix of some (N-{pi})-fair sequence, and there are no events on pi in that sequence after x. Apply requirement R (1) to the above sequence to conclude that, for any computation x and any process pi, there exists an extension z' of x such that x \perp z' and z' is j-decided for all j=1..i-1,i+1..n (i \neq j). By P 3 there is an extension z of z' in which all processes read their input. From D 2 the computation z is univalent. □

The following lemma shows that for any ROAD(t) P, every two extensions of a computation x by events only on a single process pi must have a common possible decision value. This fact should be true. Since if pi fails, then (from R(t)) the other processes will reach a decision value, and by D 2 the only possible decision value of pi is then uniquely determined. At least this value must be in all extensions of x on pi.

Definition: A set S of computations is an i-compatible set iff every two computations belonging to S are i-compatible. A set is compatible if it is i-compatible for all i=1..n.

Lemma 4: In any ROAD(t) P, for any i-input computation x the set Extensions (x,i) is a compatible set.

Proof: Let y and y' both belong to Extensions (x,i). If x is univalent then y,y' are compatible, from Lemma 1 and the fact that no computation is (i,∅)-valent for all i=1..n. For any multivalent x and pi, apply Lemma 3 to conclude that there is a univalent extension z of x such that x \perp z. From Lemma 2, both y,z and y',z are compatible. Since z is univalent, y and y' are compatible. □
In the following theorem we show that it is always possible to extend a computation to a point at which all subsequent actions of a single specific process cannot determine the final value of another process.

Definition: Computation $x$ is $j$-open w.r.t. process $p_i$ iff any computation $y \in \text{Extensions}(x,i)$ is $j$-multivalent.

**THEOREM 1:** In any ROAD($t$) $P$, for any process $p_i$ and any $j$-multivalent computation $x$, if $x$ is $i$-input then there exists an extension of $x$ which is $j$-open w.r.t. $p_i$.

*Proof:* To prove the theorem we first assume to the contrary: for some process $p_i$ and some $j$-multivalent computation $x$ where $x$ is $i$-input, there does not exist an extension of $x$ which is $j$-open w.r.t. $p_i$. Then we show that this leads to a contradiction. It follows from the assumption that for any extension $v$ of $x$ such that $p_i$ is active at $v$, there is at least one $j$-univalent computation which belongs to Extensions($v,i$); let us denote one arbitrary $j$-univalent extension of $v$ by $\Phi(v)$.

Since $x$ is $j$-multivalent, there exists an extension $z$ of $x$ ($z \neq x$) such that $z$ and $\Phi(x)$ are $j$-incompatible. Consider the extensions of $x$ which are also prefixes of $z$. Since $z$ and $\Phi(x)$ are $j$-incompatible, there exist extensions $y$ and $y'$ (of $x$) such that $\Phi(y')$ and $\Phi(y)$ are $j$-incompatible, and $y$ is a one event extension of $y'$. (Note that $y'$ and $y$ are $i$-input computations.) Therefore, $y = <y';(e,p_h)>$ for some event $(e,p_h)$. (see Figure 1.)

There are two possible cases.

**Case 1:** $p_i = p_h$. In that case $\Phi(y)$ and $\Phi(y')$ both belong to Extensions($y',i$), and hence from Lemma 4 they are compatible. However, this contradicts the assumption that $\Phi(y)$ and $\Phi(y')$ are $j$-incompatible.

**Case 2:** $p_i \neq p_h$. From Observation 1, $w = <y';(\Phi(y')-y')>$ is a computation. Since $w \in \text{Extensions}(y,i)$, it follows from Lemma 4 that $\Phi(y)$ and $w$ are compatible. However, since $w$ includes $\Phi(y')$, and $\Phi(y)$ and $\Phi(y')$ are $j$-incompatible, it follows that also $\Phi(y)$ and $w$ should be $j$-incompatible. Hence again we reach a contradiction. This completes the proof. \(\square\)

Following are two corollaries. The first follows from Theorem 1 and $P2$, and the second from Theorem 1 and $P4$.

**Corollary 1.1:** In any ROAD($t$) $P$, for any active process $p_i$ at a $j$-multivalent computation $x$, if $x$ is $i$-input then there exists a $j$-multivalent extension $y$ of $x$ such that $-(x \not\leq y)$.
Corollary 1.2: In any ROAD(t) P, for any unfulfilled event \(((\text{send}, m, p_k), p_i)\) in a \(j\)-multivalent computation \(x\), if \(x\) is \(k\)-input then there exists a \(j\)-multivalent extension \(z\) of \(x\) such that \(((\text{send}, m, p_k), p_i)\) is fulfilled in \(z\).

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

Proof: By D1(t), there exists a computation \(x\), process \(p_i\) and set of processes \(Q\), such that \(|Q| \geq n-t\), for every \(p_j \in Q\) \(x\) is \(j\)-input, \(p_i \in Q\), and \(x\) is \(i\)-multivalent. Using Corollary 1.1 and Corollary 1.2, we can construct inductively starting from the computation \(x\) a \(Q\)-fair sequence such that all the finite prefixes of that sequence are \(i\)-multivalent. This contradicts requirement R(t). □

Remark: We say that a protocol \(P'=(C',N')\) is a sub-protocol of a protocol \(P=(C,N)\) iff \(C' \subseteq C\) and \(N' \subseteq N\). It follows from Theorem 2 that no protocol has a ROAD(t) P sub-protocol.

6. An Alternative View

In this section we prove that in an asynchronous environment where processes may fail, any protocol that solves a problem in a non-trivial way has to have a computation in which (at least) one process is a decider. A decider is a process that has the ability to split the set of values it may decide on in the future. This necessary condition 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.

Let us consider the eight requirements mentioned so far. Apart from requirement D2, all the requirements capture very natural concepts: \(P1-P5\) and \(R(t)\) express the well known notions of asyn-
chronous and robust protocols respectively, while $D1(t)$ 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 expect $D2$. For later reference we call these protocols Decision($t$) Asynchronous Robust($t$) Protocols (abbr. DEAR($t$) P’s). It follows immediately from the impossibility result of Theorem 2 that DEAR($t$) P’s cannot satisfy requirement $D2$. Also, if we inspect the proof of Theorem 2 we see that requirement $D2$ is only used in the proof of Lemma 3, and Lemma 3 is only used in the proof of Lemma 4. Hence, we conclude that DEAR($t$) P’s have to satisfy the negation of Lemma 4. These observations are the base for our next theorem.

In order to state this 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 sets of values some process $p_j$ can (still) decide on in each of the resulting two computations, are disjoint. In particular when process $p_i$ is an $i$-splitter, we say that $p_i$ is a decider at a computation $x$.

**Definition:** A process $p_i$ is a $j$-splitter at a computation $x$ iff $\text{Extensions}(x,i)$ is not $j$-compatible (i.e., there exist computations $y$ and $y'$, both belonging to $\text{Extensions}(x,i)$, such that $y$ and $y'$ are $j$-incompatible).

**Definition:** A process $p_i$ is a decider at a computation $x$ iff $p_i$ is an $i$-splitter at a computation $x$.

**Theorem 3:** In any DEAR($t$) P, there exists a computation $x$ and a process $p_i$, such that $x$ is an $i$-input computation, and $p_i$ is a decider at $x$.

In order to see that this is the case, we prove two lemmas. The first lemma states that there exists a process which has the ability to split the set of values one of the processes may decide on in the future. (This process is not necessarily a decider since it may not split its own set of decision values.) As already mentioned, this lemma is merely presenting our previous result (i.e., Theorem 2) from another point of view and hence it is given without a proof. The second lemma states that no process has the ability to split the set of values another process may decide on in the future (and thus the process from the previous lemma is indeed a decider).

**Lemma 5.1:** In any DEAR($t$) P, there exists a computation $x$, process $p_i$ and process $p_j$, such that $x$ is $i$-input computation, and $p_i$ is a $j$-splitter at $x$. 
Lemma 5.2: In any DEAR(t) P, for any i-input computation x and any process p_j where i ≠ j, process p_i is not a j-splitter at x.

Proof: The proof is similar to that of Lemma 4 from the previous section. Let y and y' both belong to Extensions(x,i). Apply Lemma 3 to conclude that there is a j-univalent extension z of x such that x ≠ z. From Lemma 2, both y,z and y',z are compatible. Since z is j-univalent, y and y' are j-compatible. □

Proof of Theorem 3: From Lemma 5.1, it follows that there exist processes p_i, p_j and an i-input computation x, such that p_i is a j-splitter at x. From Lemma 5.2, it follows that in the above case, process p_i must be identical to process p_j, and hence p_i is a decider at x. □

To conclude, we have shown that the existence of a decider is a necessary condition for an asynchronous decision(t) protocol to be robust(t). An interesting open problem is to find a necessary and sufficient condition.

7. Dependent(t) Problems

In section 5, we defined for each number t, where 1 ≤ t ≤ n, a class of protocols called the class of dependent(t) protocols. We proved that any dependent(t) protocol operating in a completely asynchronous environment cannot tolerate the failure of t processes. In this section we identify problems that cannot be solved in a completely asynchronous environment, where t processes may fail. We do this by identifying those problems which are solved only by dependent(t) protocols. Hence, the impossibility of solving these problems will follow from Theorem 2.

Recall that a problem T is a mapping \( T: \overline{1} \rightarrow 2^{\overline{D}} \setminus \{\emptyset\} \) which maps each n-tuple in \( \overline{1} \) to subsets of n-tuples in \( \overline{D} \). 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 a completely asynchronous environment where t processes may fail, any protocol that solves a problem should satisfy properties P1 – P5, and the requirement R(t). Thus, we are now left with the obligation of identifying those problems which force their solutions (i.e., any protocol that solves them) to also satisfy requirements D1(t) and D2.

We now characterize the class of dependent(t) problems. Two requirements are given, and a problem is defined to be a dependent(t) problem if it satisfies these requirements. As before, the definition involves \( \text{t} \) as a parameter. Thus, for each possible value of \( \text{t} \), a different class of problems is defined.
Furthermore for any value \( t \) such that \( 1 \leq t < n \), the class of dependent(\( t \)) problems is strictly included in the class of dependent(\( t+1 \)) problems. Let \( Q \) denote a set of processes, and \( \mathbf{v} \) and \( \mathbf{v}' \) be \( n \)-tuples. We say that \( \mathbf{v} \) and \( \mathbf{v}' \) 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: \( T(H) = \bigcup_{\mathbf{d} \in H} T(\mathbf{d}) \)

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

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

**T2:** For every \( \mathbf{d} \in I \), every set of processes \( Q \) where \( |Q| = n-1 \), and every two different decision vectors \( \mathbf{d}' \) and \( \mathbf{d}'' \), if both \( \mathbf{d}' \) and \( \mathbf{d}'' \) belong to \( T(\mathbf{d}) \) 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). 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 4:** A dependent(\( t \)) problem cannot be solved in a completely asynchronous environment where \( t \) processes may fail.

**Proof:** As already explained every protocol that solves a dependent(\( t \)) problem, in a completely asynchronous environment where \( t \) processes may fail, should satisfy: \( P1-P5 \), and \( R(t) \). It follows from \( T1(t) \) that such a protocol must also satisfy \( D1(t) \). Also, it follows from \( T2 \) that the protocol satisfies \( D2 \). Hence; such a protocol is necessarily a ROA\( \text{D}(t) \) P. By applying Theorem 2 the result is proven.

For the two corollaries of Theorem 4, we use the following definitions and observations. A problem \( T: I \rightarrow 2^I - \{ \emptyset \} \) includes a problem \( T': I' \rightarrow 2^{I'} - \{ \emptyset \} \) iff (1) \( I' = I \), and (2) for every \( \mathbf{d}' \in I' \): \( T'(\mathbf{d}') \subseteq T(\mathbf{d}') \). 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 \( T': I' \rightarrow 2^{I'} - \{ \emptyset \} \) is a sub-problem of a problem \( T: I \rightarrow 2^I - \{ \emptyset \} \) iff (1) \( I' \subseteq I \), and (2) for every \( \mathbf{d}' \in I' \): \( T(\mathbf{d}') = T'(\mathbf{d}') \). 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 \( T \).
Corollary 4.1: If some sub-problem of $T$ includes only dependent($t$) problems then $T$ cannot be minimally solved in a completely asynchronous environment where $t$ processes may fail.

Corollary 4.2: If a problem $T$ has a dependent($t$) sub-problem then $T$ cannot be solved in a completely asynchronous environment where $t$ processes may fail.

Example: Consider the following (variant of the consensus) problem $T:\overline{I}\rightarrow 2^D-(\emptyset)$ where: all processes are to decide on the same value from the set $D$; $\overline{I}$ is the set of all vectors $\overline{a}$ such that $\overline{a} \in (0+1)^n$ and $|\#1-\#0| \geq t$; and there exist two input vectors $\overline{a}$ and $\overline{a'}$ such that $T(\overline{a}) \cap T(\overline{a'}) = \emptyset$. It is not difficult to see that $T$ is a dependent($t$) problem and furthermore that $T$ includes only dependent($t$) problems. From Corollary 4.1 we conclude that $T$ cannot be minimally solved in a completely asynchronous environment where $t$ processes may fail.

Remark: Consider the following modified consensus problem: Each process is to decide on a value from the set $D$, $D=\{1,2,3\}$, and it is required that all process decide "1" or all process decide on values which differ from "1". Any protocol that solves this problem, obviously is not robust(1), under the assumption that each process (regardless of its input) can initially decide on every value in $D$. The modified consensus problem is not a dependent($t$) problem (for any $t$) since it does not satisfy requirement $D2$. In the above example the values "2" and "3" are equivalent w.r.t. any process in the sense that it makes no difference on which of the two values $p_i$ decides. Based on such a notion of equivalence between decision values, it is easy to redefine the terms univalent, compatible and $Q$-equivalent, so that the impossibility result captures the above problem as well as other problems of the same nature. In the next section we show another modification of the result which also captures the above problem.

8. Other Impossibility Results

In this section we modify the results of the previous sections and then present the results by using graph-theoretic notions. One justification for this is to demonstrate the flexibility of the axiomatic proof method developed so far by showing that with minor changes in some of the definitions and proofs we gain some new results. For example the modified consensus problem mentioned at the end of Section 7 is covered. A second justification is that presenting the results from a graph-theoretic perspective may be found more intuitive and attractive to some of the readers.
The nonexistence result for ROAD(t) P actually follows from a conflict between two requirements. One is the existence of a decider which implies that a process may reach a situation where it may, so to speak, "control its own destiny". As opposed to that necessary condition we have requirement $D2$ which means that at any time a process may be forced by the group of all other processes to a situation where it has only one possible decision left. We will show that if the non-triviality requirement $D1(t)$ is strengthened and $D2$ is completely omitted, still the impossibility result holds. This is mainly due to the fact that a decider may fail at some crucial point, leaving the other processes in an uncertain state which prevents them from reaching a decision.

We first redefine the notion of valency of a computation and then define a notion which is analogous to the previous notion of an $i$–univalent computation. We use those notions to define a new class of protocols, called disconnected(t) protocols, and show that an asynchronous disconnected(t) protocol cannot be a robust(t) protocol. Let $\vec{d}=(d_1, \ldots, d_n)$ be a possible decision vector and let $W$ be a set of decision vectors.

Definition: A computation $x$ is $(W)$–valent iff (1) for every $\vec{d} \in W$, there is an extension $z$ of $x$ such that $p_i$ decides on $d_i$ in $z$ for every process $p_i$, (2) for every $\vec{d} \notin W$, there is no extension $z$ of $x$ such that $p_i$ decides on $d_i$ in $z$ for every process $p_i$.

We use the following notions. Two sets $S$ and $S'$ are conflicting iff every $\vec{d} \in S$ and $\vec{d}' \in S'$ differ by the values of at least two components. In other words, if $\vec{d} \in S \cup S'$ and $S$ and $S'$ are conflicting, then it is enough to know $n-1$ components in order to determine to which of the two sets $\vec{d}$ belongs. Two sets $S$ and $S'$ are $Q$–conflicting iff they are conflicting, and every $\vec{d} \in S$ and $\vec{d}' \in S'$ are not $Q$–equivalent.

Definition: A $(W)$–valent computation is $Q$–connected iff $W$ cannot be partitioned into two $Q$–conflicting sets. It is $Q$–disconnected otherwise.

We define a disconnected(t) protocol by simply substituting the notion of $Q$–connected for that of $i$–univalent and the notion of $Q$–disconnected for that of $i$–multivalent in the previous definition of dependent(t) protocols (see Section 4). Notice that requirement $D2$ (in the new definition) can be omitted since any computation in which $n-1$ processes have decided is $Q$–connected. Requirement $D1(t)$ means that for some $Q \geq n-t$ there is a $(W)$–valent computation such that all processes in $Q$ read their input value, and $W$ can be partitioned into two $Q$–conflicting sets.
By modification of the proof of Theorem 2, it is not difficult to show that this theorem (i.e., the impossibility result) also holds for the class of disconnected(t) protocols. As in Section 7, we next identify exactly those problems which force any protocol that solves any one of them to be a disconnected(t) protocol. Hence, the impossibility of solving these problems will follow from the previous discussion.

Definition: A problem \( T : I \rightarrow 2^D \) is a disconnected(t) problem iff it satisfies the requirement:

\[
T_1(t): \text{There exists a set of processes } Q \text{ where } |Q| \geq n - t \text{, and there exists a } Q-\text{equivalent set } H \subset I \text{ such that } T(H) \text{ can be partitioned into two } Q-\text{conflicting sets.}
\]

Before we state the next theorem we would like to describe some of the definitions using graph-theoretic notions. Define the adjacency(1) graph of \( W \) as \( G_k(W) = (W, E) \), where \( E = \{|d, \bar{d}^*| d \text{ and } \bar{d}^* \text{ differ by the values of exactly one component }\} \). One simple consequence of Theorem 4 (Section 7) can now be formulated as: A problem \( T : I \rightarrow 2^D \) cannot be solved in a completely asynchronous system where a single process may fail if there exists an input vector \( \bar{a} \in I \) such that the adjacency(1) graph of \( T(\bar{a}) \) consists of isolated vertices.

Using the notion of adjacency graph, it is easy to see that a \((W)\)-valent computation is \( Q\)-disconnected only if the adjacency(1) graph of \( W \) is disconnected. The notion of disconnected(t) problem means that \( n-t \) input values do not uniquely determine either the corresponding \( n-t \) decision values or even the connected component in the adjacency(1) graph of \( D \).

**THEOREM 5:** A disconnected(t) problem cannot be solved in a completely asynchronous system where \( t \) processes may fail.

**Proof:** The proof is similar to that of Theorem 4. Every protocol that solves a disconnected(t) problem, in a completely asynchronous system where \( t \) processes may fail, should satisfy \( P1 - P5 \), and \( Robust(t) \). It follows from \( T1(t) \) that such a protocol should also satisfy \( D1(t) \). By applying Theorem 2 (for disconnected(t) protocols) the result is proven. \( \square \)

Following is a simple consequence of Theorem 5 which extends the above consequence of Theorem 4. A problem \( T : I \rightarrow 2^D \) cannot be solved in a completely asynchronous system where a single process may fail if there exists an input vector \( \bar{a} \in I \) such that the adjacency(1) graph of \( T(\bar{a}) \) is disconnected. Also (similarly to Corollary 4.1) if a sub-problem of \( T \) includes only disconnected(t) problems then \( T \) cannot be minimally solved in a completely asynchronous environment where \( t \) processes may fail.
9. Discussion

In this paper we concentrated on a completely asynchronous model and within that model we proved impossibility results for a family of problems in the presence of multiple failures. Dolev, Dwork and Stockmeyer [DDS] took another direction. They concentrated on a fixed problem namely the consensus problem and showed (among other results) that by changing the broadcast primitives in partially synchronous models it is possible to solve the consensus problem in the presence of \( t-1 \) faulty processes but not in the presence of \( t \) faulty processes for any \( t \). Some of their results for the consensus problem can be proved to hold also for other dependent problems.

The diagram in Figure 2 describes the relations of the different classes of protocols (and problems) which are mentioned in the paper, for \(|N|=3\). We use DEPENDENT\((r)\) to denote the class of asynchronous dependent\((r)\) protocols, and ROBUST\((r)\) to denote the class of robust\((r)\) protocols. As can be seen the intersection between DEPENDENT\((r)\) and ROBUST\((r)\) is empty \((1 \leq r \leq 3)\). This follows from our main result as stated in Theorem 2. Also, as can be seen from the example mentioned in the Introduction, (1) DEPENDENT\((r)\) is strictly included in DEPENDENT\((r+1)\), (2) ROBUST\((r)\) is strictly included in ROBUST\((r-1)\), and (3) The intersection between DEPENDENT\((r)\) and ROBUST\((r-1)\) is not empty.

Nowhere in the paper have we assumed anything about the process ids, hence the result we proved holds even if all processes have distinct id's which are mutually known. It is simple to modify the presentation to allow an atomic broadcast instead of the usual send event. Also, by simply modifying

![Asynchronous Protocols Diagram](image)
property $P5$, 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).

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

References


