Information Processing Letters 26 (1987/88) 145-151
North-Holland

23 November 1987

EXTENDED IMPOSSIBILITY RESULTS FOR ASYNCHRONOUS COMPLETE NETWORKS

Shlomo MORAN * and Yaron WOLFSTAHL

Department of Computer Science, Technion, Israel Institute of Technology, Haifa, Israel 32000

Communicated by R. Wilhelm
Received 5 January 1987
Revised 13 April 1987

It is proved that a large class of distributed tasks cannot be solved in the presence of faulty processors. This class contains tasks whose unsolvability in the presence of faults is known (the consensus task and its variants, cf. Fischer et al. (1985)) as well as some new tasks (e.g., constructing a spanning tree). In particular, we introduce the notion of the *decision graph* of a task, and show that every problem whose decision graph is disconnected cannot be solved in the presence of one faulty processor, by reducing the unsolvability of this problem to the unsolvability of the consensus problem. The notion of unsolvability used here is very weak: We say that a protocol solves a given problem in spite of one faulty processor if in any execution it satisfies (i) all nonfaulty processors eventually halt, and (ii) if no processor is faulty, it solves the problem. Hence, the unsolvability of a problem in this model implies its unsolvability in other models appearing in the literature.

Keywords: Asynchronous system, consensus problem, distributed computation, fault tolerance, impossibility result, reliability

1. Introduction

The model under investigation is an asynchronous network of N processors, where each processor has a unique identity. In order to execute common tasks, the processors may communicate by exchanging messages along communic. ... links.

System reliability is a factor of crucial effect on the complexity of distributed tasks in asynchronous networks. A great amount of effort has been spent in attempts to understand the nature of that effect as well as in devising fault-tolerant protocols (cf. [1–6]).

We investigate the existence of protocols for a common type of distributed tasks, namely *decision tasks*, in the presence of faulty processors. We assume that failures may occur without any warning; a failed processor sends no messages, may not recover and its failure is undetectable (*fail-stop*).

A decision task is a function that maps each input vector, that is, a vector composed of initial values assigned to the processors, to a subset of decision (output) vectors. For example, the consensus task and the election task are typical decision tasks.

A protocol *solves a decision task in spite of one fault* if in any execution of the protocol the following holds:

(1) If no processor is faulty, the execution terminates and the eventual decision vector agrees with the task function.

(2) If at most one processor is faulty, all nonfaulty processors eventually halt.

We obtain an impossibility result for a large class of decision tasks in the presence of faults, in the sense that no protocol designed to accomplish such a task is guaranteed to ever terminate in the presence of one faulty processor. That class contains tasks whose unsolvability in the presence of faults is known (the consensus task and its variants, cf. [4]) as well as some new tasks, for example

* Part of this work was done while this author was at IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.

constructing a spanning tree. We prove our impossibility results for the very strong model of a complete network in which every processor knows the identities of all other processors. Clearly, the results also apply for weaker models in which the network is not necessarily complete, or the identities are mutually unknown.

The remainder of this paper is organized as follows: Section 2 provides the basic definitions used in the sequel. In Section 3 we prove our impossibility results. A summary and conclusions are finally given in Section 4.

2. Definitions

2.1. The model

An *asynchronous distributed system* S is a network of N ($N \geq 2$) asynchronous processors where each processor has a unique identity. The system is viewed as an undirected graph $S = (V, E)$ where $V = \{p_1, p_2, \ldots, p_N\}$ is the set of processors and E is the set of communication links connecting them. Each processor knows the identities of the processors on the other ends of its incident links. Processors communicate by sending each other messages along the communication links. Messages sent by nonfaulty processors arrive with no error in finite but unbounded and unpredictable time. Our results are proved for complete networks, in which every processor is connected to (and knows the identities of) all other processors.

2.2. Decision tasks and decision graphs

A *decision task* for a given network is a task where a decision is to be taken by the network, based on input values given to the processors. The decision consists of the decision values of all the processors, and belongs to a predefined set of possible decisions. The consensus task, where all processors are to decide on the same value, and the election task, in which exactly one processor is to decide on an 'election value', are typical decision tasks.

We now give a more precise definition of decision tasks. Let X and D be sets of *input values* and *decision values*, respectively. A *distributed decision task* T for a network of N processors is a mapping $T: X^N \to 2^{D^N}$, which maps each N-tuple in X^N to a subset of N-tuples in D^N. An *input vector* $\mathbf{x} \in X^N$ is of the form $\mathbf{x} = (x_1, x_2, \ldots, x_N)$, where x_i is the input value of processor p_i. Similarly, a *decision vector* $\mathbf{d} \in D^N$ is a vector $\mathbf{d} = (d_1, d_2, \ldots, d_N)$, where d_i is the decision value computed by processor p_i. A decision task represents, for each input vector given from some outside source, the allowable decision vectors. The set of different decision vectors in the range of T is called the *decision set* of T, and is denoted by D_T.

Remark. Our definition of decision tasks assumes that, for an input set X, every vector $\mathbf{x} \in X^N$ is a possible input to the task. In some scenarios it is assumed that the input vectors are restricted to be of some special form; for example, it is sometimes required that all the inputs values are distinct. Also, when the network is not necessarily complete, an input vector for the spanning tree task should represent an adjacency list of some graph. Our results will be generalized to such cases in the sequel.

Below, some examples of decision tasks are given.

(1) *Consensus.* In this task, the input set is arbitrary, and each input vector is mapped to the set

$$\{(1, 1, \ldots, 1), (0, 0, \ldots, 0)\},$$

the latter being the decision set. (In [4], the input set of this task was restricted to be $\{0, 1\}$; the result of Fischer et al. [4] easily generalizes to handle an arbitrary input set.)

(2) *ε-approximate agreement* [2]: In this task, the input set is the set of real numbers, and the decision vectors satisfy the following two conditions:

Agreement: The decision values of the processors are within ε of each other.

Validity: The decision value of each processor must be in the range of the initial values of the processors.

The decision set of this task is the set of all real

vectors $(d_1, d_2, \ldots, d_N)$ satisfying $|d_i - d_j| \leq \varepsilon$ for all $1 \leq i, j \leq N$.

(3) *Election:* In this task, the input set is arbitrary, and exactly one processor is to decide "1" while all other processors are to decide "0". The corresponding decision set is given as

$$\{(1, 0, 0, \ldots, 0), (0, 1, 0, 0, \ldots, 0), \ldots, (0, 0, \ldots, 0, 1)\}.$$

A well-known version of this task is to elect the processor having the largest input, where ties are resolved by the (distinct) identities of the processors. In this version, an input vector where x_i is the largest component is mapped to a single output vector where $d_i = 1$ and $d_j = 0$ for all $j \neq i$.

(4) *Ranking:* In this task, the input set is the set of integers. Each input vector is mapped to a single output vector, where the value of each component is the rank of the corresponding input component; ties are resolved by using the (distinct) identities of the processors. The corresponding decision set is

$$\{\pi \mid \pi \text{ is a permutation of } (1, 2, \ldots, N)\}.$$

(5) *Spanning tree:* No input is needed in this task, since a complete network is assumed (see the above Remark). The decision value of each processor is a set of links incident to that processor in a certain spanning tree of the network.

The following definitions play an essential roll in our proof: Let T be a decision task and let D_T be T's decision set. Two decision vectors $\vec{d}_1, \vec{d}_2 \in D_T$ are *adjacent* if $\vec{d}_1$ and $\vec{d}_2$ differ in the value of a single component only. Define the *decision graph* of T as $G_T = (D_T, E_T)$ where

$$E_T = \{(\vec{d}_1, \vec{d}_2) \mid \vec{d}_1, \vec{d}_2 \text{ are adjacent vectors in } D_T\}.$$

(A decision graph may be infinite; consider the approximate agreement task.) A task T is *disconnected* if its decision graph is not connected.

It is not hard to see that the decision graphs of all the examples above, except the ε-approximate agreement, are disconnected. (Consider, for example, the spanning tree task: In this task, any $N - 1$ decision values correspond to a set of spanning tree edges covered by $N - 1$ nodes, which must be the complete set of the spanning tree. Hence, the decision value of the N*th* processor is completely determined by these of the remaining $N - 1$ processors.) In all these examples, the decision graphs contain no edges. On the other hand, one can verify that the decision graph of the ε-approximate agreement task is connected.

3. Protocols

A *protocol* (also referred to as a *distributed algorithm* elsewhere) for a given network is a set of N programs, each associated with a single processor in the network. Within a protocol, each processor acts deterministically according to its program, which includes operations of (i) sending a message to a neighbour along a communication link, (ii) receiving a message, and (iii) processing information in its local memory. A processor that has completed its program is said to *halt*. Halting is always associated with writing a decision value.

A *configuration* of the system consists of the memory contents of each processor (including the program counter's value) together with the messages that were sent to that processor but not yet received. An *initial configuration* is one in which each processor is assigned an input value and none has started its program; note that an initial configuration is defined by an input vector. A configuration is *final* if all processors have halted; a final configuration defines a decision vector. Note that, in a nonfinal configuration, the values of some components of the eventual decision vector are not yet defined. Call such components *B-components*, and call a vector with B-components a *partial decision vector*.

Let $\vec{d}_1$ be a decision vector. A partial decision vector $\vec{d}_2$ is *extendible* to $\vec{d}_1$ if it is possible to obtain $\vec{d}_1$ by replacing all the B-components of $\vec{d}_2$ by appropriate decision values.

An *execution* of a protocol P is a sequence of events, each being either sending a message, receiving a message or doing some local computation.

Definition. A protocol P *solves* a task T if for every vector $\vec{d} \in D_T$ there is some execution of P which yields $\vec{d}$.

2.4. System unreliability

Decision tasks are solvable provided that the corresponding function is computable and the participating processors are completely reliable. Real systems, however, may be subject to processor faults. In this section we discuss the impact of unreliability on decision tasks in the presence of faults.

First, we define the notion of failure (fail-stop): A processor is *faulty* in an execution if all messages sent by that processor after a certain time are destroyed.

Since the model is fully asynchronous, in the sense that no assumptions are made about message delays, there is no way a processor can detect the failure of another processor, for there is no way to tell whether the latter is faulty or its messages are delayed. This fact is of crucial effect on the solvability of various tasks in unreliable distributed systems. For example, assume that there is an election protocol that whenever at most one processor fails, halts with exactly one processor elected (i.e., outputs "1"), and consider the following scenario. The protocol is executed, and some processor outputs "1". However, all messages sent by that processor informing the network about its election are delayed. In this case, the network cannot decide whether or not this processor already had output "1", and hence whether a leader was already elected or not. It follows that the protocol cannot guarantee that exactly one processor will be elected in the presence of a faulty processor.

Consequently, we focus our attention to a weaker form of solvability in the presence of faults, as follows. Let T be a decision task. A protocol P *solves* T *in spite of one fault* if the following hold for any execution of P:

(1) If no processor is faulty, then P solves T.

(2) If at most one processor is faulty, then all nonfaulty processors eventually halt.

A decision task is *solvable in spite of one fault* if there exists a protocol that solves that task in spite of one fault.

Note that if a task T maps all input vectors to a single decision vector, then T is trivially solvable in spite of one fault. More generally, if the decision value of each processor is a function of its input only, then the task is solvable in the presence of arbitrary many faults. Such tasks are not considered in the sequel.

3. Main results

In this section we first prove that no disconnected decision task is solvable in spite of one fault, and then we generalize that result to tasks with restricted input sets. Our proof uses a theorem proved by Fischer, Lynch and Paterson [4], restated below in terms of our model.

3.1. Theorem ([4]). *No protocol solves the consensus task in spite of one fault.*

Remark. The theorem applies here although our model slightly differs from the model of Fischer et al. [4]. In particular, Fischer et al. proved that, in the presence of a faulty processor, the system is not guaranteed to ever enter a configuration leading to just a single decision vector. This can be shown to be equivalent to our definition of unsolvability.

First, we prove two lemmas. These lemmas describe properties of decision tasks, to be used in the proof of the impossibility result.

3.2. Lemma. *Let* T *be a decision task and suppose that there exists a protocol* P *that solves* T *in spite of one fault. Consider an execution of* P *in the presence of one faulty processor. Eventually, the system will enter a configuration* C *where* $N-1$ *processors have halted, and the (partial) decision vector corresponding to* C *is extendible to a decision vector of* T.

Proof. Since the protocol solves T in spite of one fault, eventually all nonfaulty processors will halt. Suppose that the partial decision vector achieved is not extendible to a decision vector of T. As far as the nonfaulty processors are concerned, that execution is indistinguishable from another execution, in which the remaining processor is not faulty, but all messages sent by that processor are

delayed until all other processors halt. In that execution, when the remaining processor halts, the achieved decision vector is not a decision of T, which contradicts the assumption that the protocol solves T in spite of one fault. □

3.3. Lemma. *Let* T *be a decision task. Assume that* P *is a protocol that solves* T *in spite of one fault, and let* **d** *be a decision vector of* T *achieved by an execution of* P. *Then, any* $N-1$ *decision components of* **d** *uniquely determine the connected component of* **d** *in* G_T.

Proof. A set of $N-1$ decision components of **d** can be viewed as a partial decision vector with exactly one B-component. Consider two such sets, and let $\mathbf{d}_1$ and $\mathbf{d}_2$ be the corresponding partial decision vectors. Let $\mathbf{e}_1$ and $\mathbf{e}_2$ be two decision vectors to which $\mathbf{d}_1$ and $\mathbf{d}_2$, respectively, are extendible. Both $\mathbf{e}_1$ and $\mathbf{e}_2$ are adjacent to **d**. It follows that both belong to the same connected component (which is **d**'s connected component). □

We are now able to state our impossibility result. The next theorem shows that the class of disconnected decision tasks is not solvable in spite of one fault. The theorem is proved by reducing the unsolvability of any disconnected task to the unsolvability of the consensus task (whose unsolvability in spite of one fault was mentioned above).

3.4. Theorem. *No disconnected task is solvable in spite of one fault.*

Proof. Let T be a disconnected task, and let $C_1, C_2, \ldots, C_k$ be the connected components of G_T ($k \geq 2$). Assume that T is solvable in spite of one fault by a protocol P. We show that this implies that there exists a protocol P^c which solves the consensus task in spite of one fault.

The claim is proved by constructing P^c, which is based on P with the following modifications. Whenever a processor is about to halt and write a decision value in P, it broadcasts in P^c the decision value it would have if P was executed by the system (call this value a *virtual* decision value) and waits. Since P solves T in spite of one fault, eventually $N-1$ processors will broadcast their virtual decision values. Thus, each nonfaulty processor will eventually receive $N-1$ such virtual decision values that comprise, by Lemma 3.2, a partial decision vector of T.

Let ρ_i be a decision vector to which the partial decision vector achieved by processor p_i is extendible. By Lemma 3.3, the connected component C_j to which ρ_i belongs is uniquely defined, independently on the actual choice of ρ_i. Upon receiving $N-1$ virtual decision values and determining the connected component C_j, the processor will output the parity bit of j and halt. By Lemma 3.3, all processors will agree on the same component, and hence will output the same bit. Thus, P^c solves the consensus task in spite of one fault.

Since assuming that T is solvable in spite of one fault implies that there exists a protocol which solves the consensus task in spite of one fault, a contradiction to Theorem 3.1 arises. It follows that no disconnected task is solvable in spite of one fault. □

Remark. In the above proof we assumed that all processors agree on a certain order of the connected components of G_T. Such an agreement is easily achieved (with no communication) if each processor knows the identities of all other processors.

In some cases it is interesting to have impossibility results for tasks in which the inputs are restricted to be of a special form (e.g., in many cases it is assumed that all input values are distinct). Theorem 3.4 can be extended to handle this case as follows:

Let $I \subseteq X^N$ be the set of all possible inputs to a decision task T. Define the *input graph* for T similarly to the definition of a decision graph: An input vector $(x_1, x_2, \ldots, x_N) \in I$ is *adjacent* to an input vector $(y_1, y_2, \ldots, y_N) \in I$ iff they differ in the value of exactly one component. Then we have the following theorem.

3.5. Theorem. *If a task* T *has a connected input graph and a disconnected decision graph, then* T *is not solvable in spite of one fault.*

Proof. It is sufficient to prove that the consensus task, when generalized to have a connected input graph, is not solvable in spite of one fault, and then to apply the reduction of Theorem 3.4.

The proof that the consensus task, when generalized in the above manner, is not solvable in spite of one fault follows the outlines of the proof in [4], and is only sketched here: By definition of the consensus task, there are two input vectors (i.e., initial configurations) that lead to different decisions. Let x and y be these inputs. Since the input graph is connected, there is a path from x to y; on this path there must be two adjacent inputs that might lead to different decisions. From this point on, the proof is identical to the one in [4]. □

4. Conclusions

Impossibility results for a large class of decision tasks in the presence of faults have been proved. This class contains tasks whose unsolvability in the presence of faults is known (the consensus task and its variants) as well as some new tasks, such as election and constructing a spanning tree. We have shown that no protocol devised to accomplish a disconnected decision task is guaranteed to ever terminate in the presence of one faulty processor.

The model of computation which was used for the impossibility proof is very powerful, since the following assumptions have been made:

(1) The underlying graph of the network is complete, so each processor can directly communicate with any other processor.

(2) The communication links are totally reliable.

(3) The type of failure is fail-stop, leaving no room for arbitrary or malicious behaviour of faulty processors.

(4) The number of faulty processors has been limited to one.

(5) Each processor may have a different internal program.

(6) The processors' identities are mutually known.

(7) A protocol is considered to be reliable as long as the failure of a single processor does not prevent the others from halting (regardless of their decision value).

Since most reasonable models are weaker than the one discussed above, our result applies there also.

The model is fully asynchronous in the sense that both message delay and the rate of drift of the processors' internal clocks are unbounded. However, our result applies to some partially synchronous models as well. Some models in which the consensus task (hence any disconnected task) is unsolvable in spite of faults are identified in [1]. For example, it is proved in [1] that unbounded message delay and an arbitrary order of message delivery render the consensus task unsolvable in spite of one fault, even if the processors are lockstepped.

The notions of a decision graph and its connected components, as well as that of the input graph that has been introduced in this paper seem to provide a convenient tool to identify tasks which cannot be solved in the presence of a faulty processor.

The result has some pessimistic practical implications. For example, the unsolvability of the election task in the presence of faults implies that serializing access to distributed network resources is impossible in the presence of faults, since the election task may be viewed as a task where exactly one processor is allowed to access a resource.

Since no deterministic protocol is adequate for handling disconnected decision tasks in the presence of faults, a probabilistic approach seems to be the best solution in hand for these tasks. In fact, a randomized asynchronous agreement protocol like the protocol devised by Rabin [6] can be adapted to solve any disconnected decision task.

Finally, it is worth mentioning that there exist connected tasks which, apparently, are also unsolvable in the presence of faults. As an example, consider the task where the inputs are bits, and p_i $(1 \leqslant i \leqslant N)$ has to output the input bit of $p_{i\,(\mathrm{mod}\,n)+1}$.

Acknowledgment

We would like to thank the (anonymous) referee for some helpful comments, and in particular for

suggesting to extend Theorem 3.4. This suggestion led to the formulation of Theorem 3.5.

References

[1] D. Dolev, C. Dwork and L. Stockmeyer, On the minimal synchronism needed for distributed consensus, J. ACM 34 (1) (1987) 77–97.

[2] D. Dolev, N. Lynch, S. Pinter, E. Stark and W. Weihl, Reaching approximate agreement in the presence of faults, J. ACM 33 (3) (1986) 499–516.

[3] C. Dwork, N. Lynch and L. Stockmeyer, Consensus in the Presence of Partial Synchrony—A Preliminary Version, Rept. MIT/LCS/TM-270, 1984.

[4] M. Fischer, N. Lynch and M. Paterson, Impossibility of distributed consensus with one faulty processor, J. ACM 32 (2) (1985) 373–382.

[5] L. Lamport, R. Shostak and M. Pease, The Byzantine generals problem, ACM Trans. Programming Languages & Systems 4 (3) (1982) 382–401.

[6] M. Rabin, Randomized Byzantine generals, in: Proc. 24th IEEE Symp. on Foundations of Computer Science, Tucson, AZ, November 1983.