Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information Processing Letters 92 (2004) 83–87

Information Processing Letters

www.elsevier.com/locate/ipl

A constructive proof for FLP

Hagen Völzer [*]

Institute for Theoretical Computer Science, University of Lübeck, Ratzeburger Allee 160, 23538 Lübeck, Germany

Received 18 March 2004; received in revised form 3 June 2004

Available online 27 July 2004

Communicated by R. Backhouse

Abstract

We present a simple elementary proof for the result of Fischer, Lynch, and Paterson (FLP) [J. ACM 32 (2) (April 1985) 374–382] that the consensus problem cannot be solved deterministically in an asynchronous system where a single process may fail by crashing. Our proof is, in contrast to the original, constructive in its crucial lemma, showing not only that a non-terminating execution does exist but also how it can be constructed. Our proof is based on the new notion of *non-uniformity* of a configuration. Non-uniformity is different from *bivalency*, which is the central notion in the original proof as well as in proofs of related results.

Keywords: Distributed computing; Distributed systems; Fault tolerance; Concurrency; Consensus problem; Formal modelling

1. Introduction

The crash-tolerant consensus problem is one of the central paradigmatic problems in distributed computing. The fundamental result that there is no deterministic asynchronous solution to it, proven by Fischer, Lynch, and Paterson [2], has sparked a fruitful and extensive line of research. The problem can be stated as follows.

Consider a finite set of sequential processes, which communicate by sending messages over bidirectional channels. We assume that there is a channel between each pair of processes. Channels are reliable, i.e., each message that is sent is eventually received; it is not altered, multiplied, or lost during transit. However, messages may be received in a different order than in which they were sent.

Upon receipt of a single message, a process performs an atomic step, which consists of receiving the message, computing the next local state, and sending a finite set of messages. Processes are deterministic, i.e., the successor state and the messages sent depend only on the content of the received message and the state of the process when the message is received. The system is asynchronous, i.e., there are no bounds for relative speeds of processes or communication delays.[1]

[*] Tel.: +49 451 500 5314.
E-mail address: voelzer@tcs.uni-luebeck.de (H. Völzer).

[1] The FLP model seems to be more general at first sight as it also allows a process to change its state when no message arrives. This feature, however, does not increase the power of the model since a process does not gain any knowledge in an asynchronous system

Any process may stop its participation in the algorithm, in which case we say that the process *crashes*. However, we assume that at most one process crashes during an entire execution. A process that does not crash in an execution is said to be *correct* in that execution.

In the (binary) consensus problem, each process gets an input value from the set $\{0, 1\}$ and a process may irrevocably decide on a final output value such that the following three conditions are met:

- No two processes decide differently (*agreement*).
- The output value is the input value of some process (*validity*).
- Each correct process eventually decides (*termination*).

In the remainder of the paper, we formalize the model of computation and the problem (Section 2), present the new proof (Section 3), and we conclude with a short discussion.

2. The model and the problem

This section details our model of distributed computation and the consensus problem.

Let P be a finite set of *process identifiers*, S a countable set of *process states*, and M a countable set of *message values*. A *message* is a pair (p, m) where m is a message value and p is either a process identifier, denoting the *receiver* of the message, or the symbol $\perp \notin P$, signifying a message to the environment. A *configuration* c consists of a vector of process states, i.e., a mapping $\mathit{state}_c : P \to S$ and a finite multiset msgs_c of messages, which denotes the messages that are in transit in c. A *process* p consists of an *initial state* $s_p \in S$ and a *step transition function*, which assigns to each pair (m, s) of a message value m and a process state s a *follower state* and a finite set of messages (the messages to be sent by p in a step). An *algorithm* associates a process with each process identifier. We will henceforth assume that an algorithm is given and we will simply call a process identifier a process when no confusion arises.

unless it receives a message. It is easy to see that each model can be simulated by the other.

To model the consensus problem, we assume that there is initially exactly one message to each process p containing p's input value. Such a message may be interpreted as a message from the environment. Apart from these messages, channels are initially empty. Therefore, we call a configuration c of an algorithm *initial* if for each process p, $\mathit{state}_c(p) = s_p$ and msgs_c contains exactly one message $(p, \langle \text{input}, v_p\rangle)$ for each p where v_p denotes the input value of p. A process *decides* on a value v when it sends a message $(\perp, \langle \text{output}, v\rangle)$. Such a message to the environment is never received in our model.

A *step* is identified with a message (p, m). A step (p, m) is *enabled* in a configuration c if msgs_c contains the message (p, m). The step may then *occur*, resulting in a *follower configuration* c', where c' is obtained from c by removing (p, m) from msgs_c, changing p's state and adding the set of messages to msgs_c according to the step transition function associated with p. We denote this by $c \xrightarrow{p,m} c'$. An *execution from* a configuration c_0 is a finite or infinite alternating sequence $\sigma = c_0, x_1, c_1, \ldots$ of configurations and steps that starts in c_0 and ends, if it is finite, in a configuration such that for all positions i of σ, we have $c_i \xrightarrow{x_{i+1}} c_{i+1}$. A configuration c' is *reachable from* a configuration c, denoted $c \Rightarrow c'$, if there is an execution, possibly without steps, that starts in c and ends in c'. For a subset Q of processes, we write $c \overset{Q}{\Longrightarrow} c'$ if c' is reachable from c through an execution that contains only steps of processes $q \in Q$ and we write $c \overset{-Q}{\Longrightarrow} c'$ if c' is reachable from c through an execution that contains only steps of processes $q \in P \setminus Q$. Note that steps are enabled persistently, i.e., an enabled step remains enabled as long as it does not occur. The following proposition, which is easy to verify, is known as the "diamond property".

Proposition 1. *If $c \overset{Q}{\Longrightarrow} c_1$ and $c \overset{-Q}{\Longrightarrow} c_2$, then there is a configuration c' such that $c_1 \overset{-Q}{\Longrightarrow} c'$ and $c_2 \overset{Q}{\Longrightarrow} c'$.*

An *execution* of an algorithm is an execution from some initial configuration of that algorithm. A configuration is a *reachable* configuration of the algorithm if it is reachable from some initial configuration. A process p is said to be *correct* in an execution

$\sigma = c_0, x_1, c_1, \ldots$ if for each position i of σ such that c_i enables some step of p, there is a position $j > i$ such that x_j is a step of p. An execution σ is *fair* if we have for each m, each p that is correct in σ, and each position i of σ: If (p, m) is enabled in c_i then there is some $j > i$ such that $x_j = (p, m)$. Let $|P| = n$. A fair execution σ is *t-admissible*, for $t \leqslant n$, if there are at most t processes that are not correct in σ.

An algorithm *solves t-tolerant consensus* if each t-admissible execution satisfies agreement, validity, and termination.

3. The proof

First we relax the specification of consensus to obtain a problem that is solvable in our model. We say that an execution *decides v* if there is a process that decides v in that execution. Likewise, we say that a configuration *c is v-decided* if $msgs_c$ contains a message $(\bot, \langle \text{output}, v\rangle)$. We say that an algorithm solves *t-tolerant pseudo-consensus* for $t \leqslant n$ if each t-admissible execution satisfies agreement and validity and for each reachable configuration c and each set $Q \subseteq P$ with $|Q| \geqslant n - t$, there is a configuration c' that is decided for some value such that $c \overset{Q}{\Longrightarrow} c'$. It is easy to show[2] that there is an algorithm that solves t-tolerant pseudo-consensus in our model if and only if $n \geqslant 2t + 1$.

Clearly, each t-tolerant consensus algorithm is a t-tolerant pseudo-consensus algorithm. Assume for the rest of the paper that a 1-tolerant pseudo-consensus algorithm is given. To this end, we will show that each 1-tolerant pseudo-consensus algorithm has an admissible execution that does not decide and therefore, there is no 1-tolerant consensus algorithm.

[2] Ben-Or's randomized consensus algorithm [1] is a t-tolerant pseudo-consensus algorithm for $n \geqslant 2t + 1$ (replace coin flips by an assignment to a default value). For the impossibility in case of $n \leqslant 2t$, the classical argument applies that a temporary network partition may force conflicting decisions: Consider a subset P_0 of t processes that get input 0 and a subset P_1 of $n - t$ processes that get input 1. P_0 can reach a decision, which must be on 0, without any participation of P_1 and likewise P_1 can reach a decision, which must be on 1, without any participation of P_0. Proposition 1 implies that both executions can be joined, resulting in an execution with conflicting decisions.

The main idea of our proof is to consider possible decision values of subsets of $n - 1$ processes. Note that the restriction of a 1-tolerant pseudo-consensus algorithm for n processes to any subset of $n - 1$ processes is a 0-tolerant pseudo-consensus algorithm. The following definitions are based on that intuition.

We write $c \overset{p}{\Longrightarrow} c'$ for $c \overset{\{p\}}{\Longrightarrow} c'$ and $c \overset{-p}{\Longrightarrow} c'$ for $c \overset{-\{p\}}{\Longrightarrow} c'$ in the following.

Definition 1. Let c be a reachable configuration and p be a process. We say that a value $v \in \{0, 1\}$ is a *p-silent decision value* of c if there is a v-decided configuration c' such that $c \overset{-p}{\Longrightarrow} c'$. The set of all p-silent decision values of c is denoted by $\text{val}(p, c)$. A reachable configuration c is *v-uniform* if $\text{val}(p, c) = \{v\}$ for all p and *non-uniform* if it is neither 0-uniform nor 1-uniform.

Proposition 2. *Let c be a reachable configuration.*

(a) *For each process p, we have* $\text{val}(p, c) \neq \emptyset$.
(b) *If c is v-decided then it is also v-uniform.*

Proof. The claims follow directly from the definitions. □

Proposition 3. *Let $c \xrightarrow{p,m} c'$ and q be a process. We have*

(a) $p \neq q$ *implies* $\text{val}(q, c') \subseteq \text{val}(q, c)$,
(b) $p = q$ *implies* $\text{val}(q, c) \subseteq \text{val}(q, c')$, *and*
(c) $\text{val}(q, c) = \{0\}$ *implies* $\text{val}(q, c') \neq \{1\}$.

Proof. Part (a) follows directly from Definition 1. For part (b), let $v \in \text{val}(p, c)$. Then there is a v-decided configuration c_v such that $c \overset{-p}{\Longrightarrow} c_v$. From Proposition 1 follows that there is a c'' such that $c_v \overset{p}{\Longrightarrow} c''$ and $c' \overset{-p}{\Longrightarrow} c''$. The former implies that c'' is v-decided and together with the latter, we have $v \in \text{val}(p, c')$. Part (c) follows directly from parts (a) and (b). □

Lemma 1. *There is a non-uniform initial configuration.*

Proof. Let $P = \{p_0, \ldots, p_{n-1}\}$. Let c_i denote the initial configuration where process p_j has input 1 if and only if $j < i$ for $i = 0, \ldots, n$. Note that c_i and c_{i+1} differ only in the input of p_i. It follows from validity that c_0 is 0-uniform and c_n is 1-uniform. There is an index j such that c_j is 0-uniform and c_{j+1} is not 0-uniform. Because c_j is 0-uniform, we have $0 \in \text{val}(p_j, c_j)$. Therefore, there is a 0-decided configuration c such that $c_j \overset{-p_j}{\Longrightarrow} c$. Since c_j and c_{j+1} differ only in the input of p_j, we also have $c_{j+1} \overset{-p_j}{\Longrightarrow} c$ and hence $0 \in \text{val}(p_j, c_{j+1})$. It follows that c_{j+1} is not 1-uniform and therefore, it is non-uniform. □

Lemma 2. *For each non-uniform configuration c and each process p there is a configuration c' such that $c \Rightarrow c'$ and* $\text{val}(p, c') = \{0, 1\}$.

Proof. If $\text{val}(p, c) = \{0, 1\}$, we are done. Let, without loss of generality, $\text{val}(p, c) = \{0\}$. Since c is non-uniform, there is a process q such that $1 \in \text{val}(q, c)$. Then there is a 1-decided configuration c_1 such that $c \Rightarrow c_1$. Because c_1 is 1-decided, we have $\text{val}(p, c_1) = \{1\}$. Due to Proposition 3(c), there is a configuration c' such that $c \Rightarrow c' \Rightarrow c_1$ and $\text{val}(p, c') = \{0, 1\}$, which is what we wanted to show. □

We are now ready to prove the main theorem, where we use the following definition. Let $\sigma = c_0, x_1, c_1, \ldots, c_k$ be a finite execution and (p, m) be a step that is enabled in c_k. The *enabling time* of (p, m) in σ is the smallest position l of σ such that c_j enables (p, m) for all $j = l, \ldots, k$ and $x_j \neq (p, m)$ for all $j = l + 1, \ldots, k$.

Theorem 1. *Each* 1*-tolerant pseudo-consensus algorithm has a* 0*-admissible execution that does not decide.*

Proof. To construct the non-deciding execution, start in a non-uniform initial configuration, which exists according to Lemma 1. Then, repeat the following as long as possible. Take an enabled step (p, m) with the minimal enabling time and extend, according to Lemma 2, to a configuration c such that $\text{val}(p, c) = \{0, 1\}$. Configuration c is therefore non-uniform. If (p, m) is enabled in c let (p, m) occur and call the resulting configuration c'. Due to Proposition 3(b), we have $\text{val}(p, c') = \{0, 1\}$ and c' is hence non-uniform.

We obtain a fair execution where all processes are correct and that is always eventually[3] non-uniform and hence does not decide. □

4. Conclusion

Consensus is a coordination problem where the consistent choice of a final value has to be coordinated. When no randomization is available, coordination can be achieved only through synchronization. The FLP result shows that an asynchronous system cannot provide the required synchronization.

In the original proof [2], the central lemma, which states that every enabled step (p, m) can eventually be executed without determining a decision value, is proven by way of contradiction. We have shown how such an extension can be constructively obtained: If v is not a p-silent decision value, then drive the execution towards a decision on v until both values are p-silent decision values, which assures that the occurrence of (p, m) preserves both decision values.

The original proof and other proofs of related results are based on the notion of *bivalency*, where a configuration is *bivalent* if a 0-deciding configuration as well as a 1-deciding configuration can be reached from it. We introduced the notion of non-uniformity which gives rise to a simpler proof. Non-uniformity is weaker than bivalency, i.e., non-uniformity implies bivalency but bivalency does not imply non-uniformity. We hope that the new notion can help to simplify other proofs that are based on bivalency.

Acknowledgement

We would like to thank an anonymous referee who helped to improve the presentation of this material.

[3] It is easy to show that our construction can also assure that the execution is always non-uniform.

References

[1] M. Ben-Or, Another advantage of free choice: Completely asynchronous agreement protocols, in: Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, ACM, 1983, pp. 27–30.

[2] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (April 1985) 374–382.