Information Processing Letters 115 (2015) 562–569

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Self-stabilizing distributed algorithm for local mutual inclusion

Hirotsugu Kakugawa[1]

Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan

ARTICLE INFO

Article history:
Received 19 November 2013
Received in revised form 27 January 2015
Accepted 28 January 2015
Available online 3 February 2015
Communicated by M. Yamashita

Keywords:
Distributed algorithm
Mutual inclusion
Mutual exclusion
Process synchronization
Self-stabilization

ABSTRACT

Local mutual inclusion is a process synchronization problem where, for each process, at least one of its processes and neighbors must be in the critical section. We propose a self-stabilizing distributed solution to the local mutual inclusion problem. Convergence time of the proposed algorithm is one round under the weakly fair distributed daemon.

1. Introduction

The distributed mutual inclusion problem is the complement of the distributed mutual exclusion problem [17,16]; unlike mutual exclusion, where at most one process is in the critical section (CS), mutual inclusion places *at least* one process in the CS. In [11], a solution for only two processes is proposed. This solution involves semaphores, which are variables or abstract data types that are used by multiple processes for access control.

We propose a self-stabilizing and distributed solution to the local mutual inclusion problem. The mutual inclusion problem is *local* if, for each process P, at least one of P and its neighbors are in CS. It is *global* if, at least one process in the system is in CS. The global mutual inclusion problem is a special case of the local mutual inclusion problem because they are the same when the network topology is complete. Self-stabilization is a class of fault-tolerant algorithms for transient faults [6,23]. The main motivation for studying the distributed local mutual inclusion problem is the theoretical formulation of the handover of cluster heads in sensor networks. Processes may have to change their role (cluster head or ordinary) to equalize energy consumption; these changes have to be done in such a way that the clustering condition is ensured, i.e., each ordinary process is at a distance of at most one cluster head. Suppose a process is in the CS if and only if it is a cluster head. In this case, local mutual inclusion maintains a clustering condition; that is, each process is a cluster head or at least one of its neighbors is a cluster head. The self-stabilizing property implies that the proposed solution is fault-tolerant to transient faults and adaptive to topology changes. Moreover, a global reset is not necessary when a distributed system starts.

This paper is organized as follows. In Section 2, we give several definitions and problem statements. In Section 3, a self-stabilization solution to the local mutual inclusion problem is presented. We prove the correctness of this proposed algorithm and analyze its performance in Section 4. In Section 5, we review related works and discuss

E-mail address: kakugawa@ist.osaka-u.ac.jp.

[1] This work is supported in part by Grant-in-Aid for Scientific Research (C)24500039 of JSPS.

http://dx.doi.org/10.1016/j.ipl.2015.01.008
0020-0190/© 2015 Elsevier B.V. All rights reserved.

our results. In Section 6, we conclude this paper and discuss future research.

2. Preliminaries

Let n be the number of processes, $V = (P_1, P_2, \dots, P_n)$ be a set of processes, and $E \subseteq V \times V$ be a set of communication links in a distributed system. The topology of the distributed system can be represented as the graph $G = (V, E)$. Assume that G is connected, simple, and undirected. For each communication link $(P_i, P_j) \in E$, we say that P_j is a *neighbor* of P_i. Since a communication link is bidirectional, $(P_i, P_j) \in E$ if and only if $(P_j, P_i) \in E$ for each $P_i, P_j \in V$. The set of neighbors of P_i is denoted by N_i, and a local variable x at process P_i is denoted by x_i. A set of local variables defines the local state of a process. Let Q_i be the local state (a vector of all local variables) of process $P_i \in V$. A vector of local states $(Q_1, Q_2, \dots, Q_n)$ of all processes forms a *configuration* (global state) of a distributed system; the set of all configurations is denoted by Γ.

An algorithm of each process P_i is given by a finite set of guarded commands:

```
Rule 1:  if $Grd_1$ then $Act_1$
Rule 2:  if $Grd_2$ then $Act_2$
         ⋮
Rule $L$:  if $Grd_L$ then $Act_L$.
```

Each Grd_ℓ ($\ell = 1, 2, \dots, L$) is called a *guard*, and it is a predicate on P_i's local state and the local states of its neighbors. For communication model, we assume that each process can read the local state of its neighbors, which is called the *state-reading model*. Although a process can read local states of neighbors, it can only update its local state. We say that P_i is *enabled* in configuration γ if and only if at least one guard of P_i is true in γ. If P_i is not enabled, we say that P_i is *disabled*.

Each Act_ℓ is called an *action* or *move*, which updates the local state of P_i. The next local state is computed from the current local state of P_i and those of its neighbors.

An atomic step of each process P_i consists of the following three internal sub-steps known as the *composite atomicity* model: read local states of neighbors and evaluate guards, execute an action that is associated with a true guard (if any exist), and update its local state.

For the execution model, we assume the *distributed daemon*, which is often assumed in the literature of self-stabilizing distributed algorithms. At each step, the distributed daemon arbitrarily selects a non-empty subset of enabled processes, and the selected processes execute their actions simultaneously. We also assume the distributed daemon is *weakly fair*; that is, a process which is continuously enabled is eventually selected to take an action by the distributed daemon.

2.1. Mutual inclusion

Assume that each process $P_i \in V$ virtually maintains a local variable $state_i \in \{\mathsf{InCS}, \mathsf{NonCS}\}$, and assume the initial value of $state_i$ is InCS. The behavior of each process P_i is as follows. Note that we assume that P_i eventually invokes Entry-Sequence when it is in the NonCS state.

```
/* InCS */
while true {
  Exit-Sequence;
  /* NonCS */
  Entry-Sequence;
  /* InCS */
}
```

Definition 1 *(Local mutual inclusion).* A protocol $\mathcal{P}$ solves the local mutual inclusion problem if and only if the following two conditions hold:

Safety: For each process $P_i \in V$, at least one P_i, or its neighbor $P_j \in N_i$, is in the InCS state.
Liveness: Each process $P_i \in V$ enters the NonCS and InCS states alternately infinitely often.

The goal of local mutual inclusion is to design a protocol for Exit-Sequence and Entry-Sequence that meets the definition. Note that local mutual inclusion defined above is local in the sense that for each P_i, at least one process among $N_i \cup \{P_i\}$ must be in the InCS state. In the special case that G is a complete network, this problem is referred to as *global* mutual inclusion. In this paper, we propose an algorithm for local mutual inclusion.

2.2. Self-stabilization

The self-stabilization property is defined as the ability to converge to a correct system operation in finite time from an arbitrary initial configuration. Let $S = (\Gamma, F, \to)$, where Γ is the finite set of all configurations, F is the predicate on a sequence of configurations,[2] and $\to$ is a relation on $\Gamma \times \Gamma$. $S = (\Gamma, F, \to)$ can be viewed as a transition system defined by a given network topology and algorithm. For any configuration $\gamma \in \Gamma$, let $\gamma' \in \Gamma$ be any configuration that follows γ by a single step of execution. We denote this transition relation by $\gamma \to \gamma'$. We say that $\gamma \overset{*}{\to} \gamma'$ if and only if $\gamma_0 (= \gamma) \to \gamma_1, \gamma_1 \to \gamma_2, \dots,$ and $\gamma_{t-1} \to \gamma_t (= \gamma')$ for some $t \geq 0$.

Definition 2. For any configuration γ_0, a *computation* $e(\gamma_0)$ starting from γ_0 is a maximal (possibly infinite) sequence of configurations $e(\gamma_0) = \gamma_0, \gamma_1, \gamma_2, \dots,$ where $\gamma_t \to \gamma_{t+1}$ for each $t \geq 0$. A computation is said to be maximal if (1) it is infinite, or (2) it is finite and no process is enabled in the last configuration.

If the initial configuration γ_0 is clear from the context, the computation is denoted by e instead of $e(\gamma_0)$.

[2] For example, for a token circulation algorithm, F must be true when a token is circulated, and there is only one token. Hence, for such a non-silent algorithm, the predicate F must be defined over a sequence of configurations rather than a single configuration.