# The Fuzzy Philosophers

Shing-Tsaan Huang

Department of Computer Science and Information Engineering
National Central University
Chung-Li, Taiwan 32054, R.O.C.
E-mail: sthuang@csie.ncu.edu.tw

Consider a network of nodes; each node represents a philosopher; links represent the neighboring relationship among the philosophers. Every philosopher enjoys singing so much that once getting the chance, he always sings a song within a finite delay. This paper proposes a protocol for the philosophers to follow. The protocol guarantees the following requirements: (1) No two neighboring philosophers sing songs simultaneously. (2) Along any infinite time period, each philosopher gets his chances to sing infinitely often. Following the protocol, each philosopher uses only one bit to memorize his state.
Sometimes the philosophers may be fuzzy enough to forget the state. So, a self-stabilizing version of the protocol is also proposed to cope with this problem. However, the philosophers may need additional bits to memorize their states.

## 1. Introduction

Consider a network of nodes; each node represents a philosopher; links represent the neighboring relationship among the philosophers. This paper proposes a protocol for the philosophers to follow. The protocol guarantees the following two requirements: (1) No two neighboring philosophers sing songs simultaneously. (2) Along any infinite time period, each philosopher gets his chances to sing infinitely often. Following the protocol, each philosopher only uses a boolean variable to memorize his state.
Sometimes the philosophers may be fuzzy enough to forget the state. The fuzzy behavior of the philosophers is modeled as *transient faults*. A transient fault may perturb the values of the variables of a program but not the constants and the program code. To cope with all kinds of possible transient faults, Dijkstra [3] introduced the *self-stabilizing* (SS in short) concept into computer systems. Provided that no more transient faults may occur afterwards, an SS system must be able to stabilize eventually to states which fulfil the desired requirements no matter what current state it is.
Singing a song by the philosophers can be modeled as executing the *critical section* (CS in short). Then, the formulated problem is closely related to the dinning philosophers by Dijkstra [4] and the drinking philosophers by Chandy and Misra [2] although the dinning philosophers and the drinking philosophers do not handle

transient faults. That no two neighboring philosophers are allowed to execute the CS simultaneously is the common requirement. The major issue faced in the philosophers in fulfilling the requirement is the symmetry problem. It would be impossible to have a deterministic solution if the system is in a state of which no node is distinguishable from the others. Here in this paper, a simple and elegant approach is proposed which allows a node use only one bit to resolve the conflicts. The result should be interesting to those who might design distributed protocols to resolve the conflicts among the requests from neighboring processes.

There are two versions of the proposed protocol: *A-protocol* and *B-protocol*. A-protocol has the SS property if the network is acyclic, but not otherwise. B-protocol can cope with the transient faults; i.e., it is an SS protocol. Provided that the philosophers are not fuzzy any more, B-protocol eventually guarantees the two requirements. However, the philosophers may need more boolean variables to memorized their states.

An SS protocol is usually presented in rules. Each rule has two parts: the *guard* and the *action*. The guard is a boolean function of the states of the node and its neighbors. If the guard is true, its action is said to be *enabled* and can then be executed. In proving the correctness of an SS protocol, the following three assumptions may be considered:

(1). *Serial execution*: Enabled actions are executed one at a time.

(2). *Concurrent execution*: Any nonempty subset of enabled actions are executed all at a time.

(3) *Distributed execution*: A node may read the states of its neighbors at some different times and evaluate its guards and execute the enabled actions at a later moment.

A distributed-correct protocol is also concurrent-correct, in turn, is also serial-correct; but, not vice versa. Because it is easier to design and prove serial protocols, most of the SS protocols[3] are design in such a way.

The result reported in this paper is inspired by *the alternator* studied by Gouda and Haddix [5]. One major difference between their result and the current one is that their protocol supports correct concurrent execution of serial-correct SS protocols, whereas the proposed B-protocol supports not only correct concurrent execution but also correct distributed execution. Correct-distributed execution is commonly believed more difficult.

A rule is said to be *non-interfering* if once it is enabled, it remains so until the action part is executed. It has been shown that a serial-correct protocol is also distributed-correct provided that its rules are non-interfering [1]. The non-interfering property of the rules makes the proposed B-protocol can support correct distributed execution for the serial-correct SS protocols. Other attempts made to support correct distributed execution for serial-correct SS protocols can also be found in [6],[7].

The rest of the paper is organized as follows. Section 2 presents A-protocol. Next, Section 3 gives its correctness proof. B-protocol and its correctness discussion are then presented in Section 4. The efficiency of A-protocol is discussed in Section 5.

## 2. A-protocol

The first issue we face is the symmetry problem. To solve the problem, in A-protocol, each link is assigned a *static direction* such that the directed network is acyclic. The directed link is then called the *base edge* and is denoted by (B→). The directed networked induced by the base edges is called the *Bnetwork*. Note that the

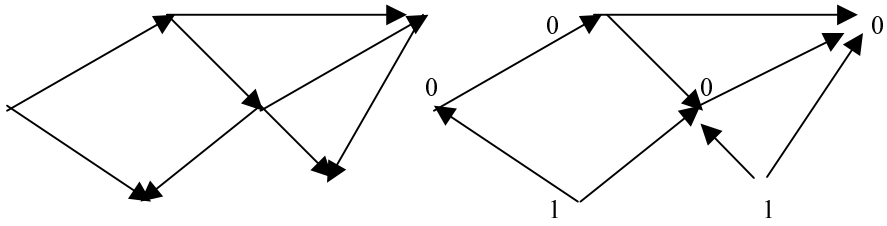
Bnetwork is static in the sense that all the directions of its edges are fixed. Hence, the Bnetwork is always acyclic.

Associated with each link, there is another edge called the *control edge*, denoted by (C→). The direction of the control edge is dynamically controlled by two control bits maintained by the two nodes incident to the edge, respectively, via the following four rules:

0(B→)0 then (C→)
0(B→)1 then (C←)
1(B→)0 then (C←)
1(B→)1 then (C→)

Let the control bit maintained by node i be denoted as C.i. For two neighboring nodes i and j, the rules imply that if C.i ⊕ C.j = 1, then the control edge has the reversed direction of the base edge, otherwise they have the same direction. Where ⊕ is the exclusive OR operator. The directed network induced by the control edges is called the *Cnetwork*.

According to the four rules, a node can reverse all the directions of its adjacent control edges simply by reversing its control bit. Figure 1 gives an example for the Bnetwork and the Cnetwork. The following A-protocol is a direct consequence of this surprisingly simple result.




(a) Bnetwork (b) Cnetwork

Figure 1. Example for the base network and the control network.

Let Csink.i (or Csource,i, respectively) denote that all the control edges of node i are incoming to (or outgoing from, respectively) i. A-protocol consists of one guarded rule only.

[R0.A] Csink.i → Execute CS, C.i := ¬C.i.

The idea behind A-protocol is very simple. The control edge is used as an arbitrator to decide which one of the two nodes incident to the edge has the priority to execute the CS: the one pointed to has the priority. Csink.i implies that all the neighbors of i agree with that node i has the priority. After executing the CS, node i yields the priority to all its neighbors by reversing its control bit. With common knowledge in the mutual exclusion field [2], A-protocol obviously has the *safety* property, i.e., no two neighboring nodes execute the CS simultaneously.

[R0.A] is *non-interfering*: once the guard is true, it remains true until the action part is executed. The non-interfering property of the rule makes A-protocol distributed-

correct, provided that it can be proved serial-correct [1]. Therefore, the correctness proof in the next section only considers serial execution.

## 3. Correctness of A-protocol

We prove A-protocol correct by showing that it has the following two properties.

[P0]: (Safety Property) No two neighboring nodes execute the CS simultaneously.

[P1]: (Fairness Property) Along any infinite computation, each node executes the CS infinitely often.

As discussed in the previous section, the following Theorem 1 is true.

Theorem 1: A-protocol has the property [P0].

In order to show that A-protocol also has the property [P1], we show that A-protocol is deadlock-free, first. Let all the control bits maintained by the nodes be initialized as zero and assume that the system faces no transient faults. Later, in B-protocol, we will discuss how to handle the transient faults. Under this assumption, we have the following invariant.

[I0] The Cnetwork is acyclic.

Lemma 1: [I0] is an invariant.

Proof: First, [I0] is true at the time when the network is initialized. This is because the Cnetwork is exactly the Bnetwork at the beginning.

Secondly, if [I0] is true before a system state transition, it is also true after the transition. Note that a node changes from a Csink node to a Csource node when it executes the action part of the rule. This is because all the control edges of the node reverse their direction by the action part of the rule. Also, it's not hard to see that an acyclic network remains acyclic if some sink node is replaced with a source node.

All those together implies that [I0] is an invariant. This ends the proof.

Lemma 2: A-protocol is deadlock-free.

Proof: By Lemma 1, the Cnetwork is always acyclic; hence, at any state there exists at least one Csink node, which is enabled. This ends the proof.

Theorem 2: A-protocol has the [P1] property.

Proof: [R0.A] is non-interfering; hence, an enabled node executes the rule eventually. Then, by Lemma 2, along any infinite computation, some node, say node i, must execute the CS infinitely often. By the rule, between two successive CS steps of node i, all its neighbors must execute the CS once. Hence, all the neighbors of node i must execute the CS infinitely often along the computation. Then, because the network is finite, this theorem is proved.

We have proved the correctness of A-protocol under the assumption that no transient faults may occur. However, when transient faults are taken into consideration, [I0] is no longer an invariant. To see this, consider a three-node ring with the following Bnetwork configuration: $i(B\rightarrow)j(B\rightarrow)k(B\leftarrow)i$. At some moment, the Cnetwork configuration may be as $0(C\rightarrow)0(C\leftarrow)1(C\rightarrow)0$. Then, a transient fault may perturb C.i

changing its value from 0 to 1 and make the configuration as 1(C←)0(C←)1(C←)1. A cycle exists. In the next section, B-protocol is modified from A-protocol to cope with the transient faults. Note that A-protocol has the SS property if the original network is acyclic; this is because in such a case the invariant [I0] is valid even after the transient faults.

4. B-protocol

The idea behind B-protocol is to color the links of the network into different colors: color-1, color-2, …, color-m. The subnetwork induced by links with color-x is called the Cx-subnetwork. The coloring must be carried out in such a way that each of the C1-subnetwork, C2-subnetwork, …, and Cm-subnetwork is acyclic but may be disconnected. Here we assume the colors are initially given.

According to the colors of the links, nodes are classified into non-mutually-exclusive, different color sets. A node is said to belong to Cx-set (or said to be a Cx node) if the node is incident to at least one link with color-x. Note that a node may belong to several different color sets. As an example, one may color a mesh with two colors: the vertical links with color-1, and the horizontal links with color-2. In such a coloring, each node belongs to two color sets.

In B-protocol, if node i is a Cx node then i maintains a control bit for those links with color-x, denoted as Cx.i, to control the control edges over those links. Hence, for a node belongs to k different color sets, k control bits are needed. Similar to A-protocol, associated with each link, there is a base edge. However, the direction of the base edges can be arbitrary. The requirement that Bnetwork induced by the base edges is acyclic is no longer necessary. The requirement is needed in A-protocol because the invariant [I0] must be initially true. The direction of the control edge over a link with color-x is decided by the direction of the associated base edge and the four rules in the same way as in A-protocol.

B-protocol consists of two rules. The notation Cx-sink in the rules is corresponding to Cx-subnetwork.

[R0.B]: $\forall$Cx: node i $\in$ Cx-set: Cx-sink.i $\rightarrow$ Execute CS, Cx.i := $\neg$Cx.i.

[R1.B]: $\exists$Cx, Cy: x < y, node i $\in$ Cx-set, node i $\in$ Cy-set: $\neg$Cx-sink.i $\wedge$ Cy-sink.i
$\rightarrow$ Cy.i := $\neg$Cy.i.

Rule [R0.B] guarantees that B-protocol has the safety property [P0] because its guard guarantees that all the control edges incident to node i point to i. The rules imply that each node waits for executing CS by waiting to hold needed sink status of the control subnetworks one by one from lower color to higher color. By [R1.B], when a node does not hold the needed sink status of a lower color control subnetwork, it does not keep the sink status of a higher color control subnetwork to avoid deadlock.

A-protocol is proved serial-correct. It is also distributed-correct because the only rule [R0.A] is non-interfering. This is not the case for B-protocol because [R1.B] is not non-interfering. The guard of [R1.B] may change from true to false if the action part of it does not execute in time. However, what we really care is the execution of the CS by the nodes, that is, the action part of rule [R0.B]. Rule [R0.B] is obviously non-interfering. Therefore, we conclude that B-protocol is also distributed-correct as long as the two properties [P0] and [P1] are the only concerns.

B-protocol can support correct distributed execution of the serial-correct application protocol in a very simple way. The rules of the application protocol are simply attached into the CS part of B-protocol. The node holding the CS privilege according to B-protocol then executes the rules of the application protocol.

## 5. Efficiency of A-protocol

This section discusses the efficiency of A-protocol. We are unable to derive good results regarding the efficiency of B-protocol. In the discussion, a *maximal concurrent execution* of A-protocol is assumed as in [5]. That is, the nodes are executed in locked steps; in each step(y,z), which bring the system from state y to state z, all enabled actions at state y are executed in the step.

A-protocol assumes no transient faults. Hence, the Cnetwork is acyclic. A *Cpath* is defined as a directed path from a non-sink node (the *head* node of the Cpath) to a sink node (the *tail* node of the Cpath) in the Cnetwork. Note that the head node is not necessary a source node, also from a non-sink node, many Cpaths may exist. Hence, a Cpath may include many shorter Cpaths. For example, Cpath (h, i, j, …,w) includes Cpath (j, …,w).

The *length* of a Cpath is defined as the number of edges in it, which can only become shorter because the head is fixed and the tail can only shrink. The maximum length of all the Cpaths from a node is the lowest possible number of steps that the node needs to wait for its turn to execute the CS. Therefore, the maximum length of all directed paths in the network, denoted as Xlength of the network, is used as the metric in discussing the efficiency.

Lemma 3. In each step(y,z), an existing Cpath at state y becomes one edge shorter or disappears at state z.

Proof: At state y, except the tail node, which is a sink, all other nodes, including the head node and the middle nodes, of the path are not enabled, and so they remain in the path at state z. Whereas, the tail node is enabled at state y, and hence its action part is executed in the step and reverses the direction of all the control edges incident to it. In other words, the tail node of the path at state y is no longer part of it at state z. This ends the proof.

Note that two or more Cpaths with the same head node may merge into one when they are getting shorter and shorter. For example, Cpath(h, …,u,v,w) and Cpath(h, …u,s) merge into one as Cpath(h, …u,v) at the next state. Also, a new Cpath may be created because a sink node becomes a non-sink node at the next state.

Lemma 4. In each step(y,z), a newly created Cpath has length one, or can only be as long as some Cpath existing in state y.

Proof: Let Csink.i at state y. Node i becomes a non-sink node at state z. Then, (i,j) may be a new Cpath of length one. Or, (i,j, …h) may be a new Cpath because Cpath(j, …h,k) exists at state y; both have the same length. This ends the proof.

Lemma 5. The Xlength of the Cnetwork is non-increasing.

Lemma 5 is a direct consequence of Lemmas 3 and 4. By Lemma 5, the Xlength of the Cnetwork can only become smaller during the computation. Recall that at the beginning, the Cnetwork is exactly the Bnetwork; therefore, A-protocol can be made very efficient by assigning direction of the links of the network in such a way that the Xlength is made as small as possible. For example, in a ring network, the Xlength can be at most two.
A-protocol is an SS protocol when it applies on an acyclic network (viz. tree network) as mentioned before. When the network is acyclic, A-protocol is very efficient according to the following Theorem 3.

Theorem 3. The Xlength of the Cnetwork stabilizes to one when A-protocol applies on a finite tree network.
Proof: Eventually, any newly created Cpath in each step(y,z) can only be of length one at state z. This is because the Cnetwork on a finite tree network is finite and acyclic; the creation of a new Cpath of the form (i,j, …h) eventually becomes impossible.
When this happens, each step afterwards, existing Cpaths become one edge shorter and newly created Cpaths are of length one. Hence, the Xlength of the Cnetwork stabilizes to one. This ends the proof.

Theorem 3 implies that when A-protocol applies on a tree network, the system stabilizes to states in which a node can execute the CS once every two steps.

Acknowledgement:
This research was supported in part by the National Science Council of the Republic of China under the Contract NSC 89-2213-E-007-043.

References:
1. Brown, G. M., Gouda, M. G., and Wu, C. L.: Token systems that stabilize. IEEE Transaction on Computers, Vol. 38, No. 6 (1989) 845-852.
2. Chany, K. M. and Misra, J.: The drinking philosophers problem. ACM Transaction on Programming Languages and Systems, Vol. 6, No. 4, Oct. (1984) 632-646.
3. Dijkstra, E. W.: Self stabilizing systems in spite of distributed control. Communications of the ACM, Vol. 17, No. 6 (1974) 643-644.
4. Dijkstra, E. W.: Hierarchical ordering of sequential processes. In Operating Systems Techniques, Hoare, C.A. R. and Perrott, R.H., Eds., Academic Press, New York (1972).
5. Gouda, M. and Haddix, F.: The alternator. Proceedings of the 1999 Workshop on Self-Stabilizing Systems (WSS-99) 48-53.
6. Huang, S.T., Wuu, L.C., and Tsai, M. S.: Distributed execution model for self-stabilizing systems. Proceedings of the 14th International Conference of Distributed Computing Systems. (ICDCS-94) (1994) 432-439.
7. Mizuno, M., Nesterenko, M., and Kakugawa, H.: Lock based self-stabilizing distributed mutual exclusion algorithms. Proceedings of the 16th International Conference of Distributed Computing Systems. (ICDCS-96) (1996) 708-716.