A Stabilizing Deactivation/Reactivation Protocol

Mehmet Hakan Karaata and Mohamed G. Gouda

Abstract—Consider a distributed system that delivers a set of services (such as message routing, maintenance of a global invariant, leader election, mutual exclusion, and so forth) to a distributed application. Such a system often provides its services at all times, regardless of whether or not these services are in demand at any given time. This leads to wasteful use of system resources. In this paper, we propose a novel stabilizing protocol for deactivating the system services in the absence of demand and reactivating the services upon demand. The proposed protocol is simple enough. When a process needs a service, it periodically sends messages that reach every other process in the system and causes every process to reactivate the service. For this purpose, only a single-type message carrying no information is sent in the system. When no process needs the service, the sending of messages is stopped, causing every process to deactivate the service. The proposed system has many applications in mobile and sensor networks.

Index Terms—Deactivation/reactivation, fault tolerance, network protocols, sensor networks, stabilization.

1 Introduction

Consider a distributed system delivering a set of services to a distributed application such as message routing, maintenance of a global invariant, leader election, mutual exclusion, and so forth. Assume that, in such a system, each process has two modes of operation, namely, *active* and *deactivated*. A process in the active mode performs all of the regular process activities, whereas, in the deactivated mode, the process suspends all regular activities except for the reactivation activities. A reactivation activity involves either sensing some activity in the environment of the process or the receipt of a message that informs the process that another process in the system is active. Most systems often keep all processes in the active mode at all times, regardless of whether or not system services are in demand at any given time. This leads to the wasteful use of system resources. Ideally, these services are provided on demand, that is, all system processes delivering the services are activated in the presence of these demands and deactivated in the absence of demands. A *deactivation/reactivation protocol* ensures that two conditions are satisfied about the status of the system processes. First, after the system enters a system state where no process in the system experiences local reactivation activity, all system processes are in the deactivated mode in a bounded time and the system processes remain deactivated as long as no system process is experiencing any local reactivation activity. Second, after the system enters a system state where there exists at least one system process experiencing local reactivation activity, all system processes enter the active mode in bounded time and the system processes remain active as long as there exists at least one system process experiencing local reactivation activity. Service deactivation/reactivation is especially critical in sensor networks for conserving scarce power. Sensor nodes have a special mode called "inactive" in which the node consumes a fraction of the regular power consumption.

Deactivation/reactivation protocols have a wide range of applications such as leader election and on-demand routing in distributed systems. For example, consider a distributed application requiring leader election for some of its tasks for a limited duration of time. In this system, when the leader election is required by the distributed application, one or more of the distributed application processes indicate the demand for leader election to their corresponding leader election subsystem processes. Then, upon detecting that the distributed application requires leader election, a subset of the leader election subsystem processes spontaneously wakes up and starts the leader election algorithm by activating all leader election subsystem processes. With the participation of all subsystem processes, the leader election algorithm elects a leader in the system. Upon completion of the leader election, the distributed application completes its task requiring the leader election and indicates to all of the leader election subsystem processes the absence of demand for leader election. Then, no process of the leader election subsystem observes the demand for the leader election. Consecutively, all of the leader election processes are deactivated. In addition, to minimize the control overhead in ad hoc mobile and sensor networks, on-demand routing tables for interprocess communication are built only as desired by the source nodes and maintained as long as they are on demand by sources. For this purpose, all or some of the system processes are activated to build the routing tables and to perform the routing. Then, when no source process remains in the system (or in a locality), the routing tables are destroyed and all of the system processes are deactivated.

On-demand routing protocols have been proposed for sensor, mobile, and radio networks. For instance, Perkins and Royer designed an ad hoc on-demand distance vector routing algorithm for ad hoc mobile networks, implementing both unicast and multicast routing [1]. Their algorithm builds routes between nodes only as desired by source nodes and

- *M.H. Karaata is with the Department of Computer Engineering, Kuwait University, PO Box 5969, Safat 13060, Kuwait. E-mail: karaata@eng.kuniv.edu.kw.*
- *M.G. Gouda is with the Department of Computer Science, University of Texas at Austin, 1 University Station (C0500), Austin, TX 78712-0233. E-mail: gouda@cs.utexas.edu.*

Manuscript received 7 Dec. 2005; revised 7 June 2006; accepted 20 Nov. 2006; published online 2 Apr. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0434-1205.
Digital Object Identifier no. 10.1109/TC.2007.1048.

maintains these routes as long as they are in demand by the sources. In a related work, Hu et al. propose a secure on-demand routing protocol for ad hoc networks [2]. In [3], Kravets and Zheng propose an extensible on-demand power management framework for ad hoc networks that adapts to traffic load.

Stabilization is a desirable property for a deactivation/reactivation protocol to eliminate the system initialization and withstand *transient faults*. A stabilizing system guarantees that, regardless of the current configuration, the system reaches a legal state in a bounded number of steps and the system state remains legal thereafter. Due to this attribute, stabilizing algorithms are able to withstand transient failures. We view a fault that perturbs the state of the system but not the program as a transient fault. Due to these features, devising stabilizing distributed sensor and mobile network protocols is desirable.

In the area of stabilization, the concept of stabilizing deactivation/reactivation is related to the *silent stabilization* paradigm of Dolev et al. [4]. A silent stabilizing system is guaranteed to enter a state where all of the actions of the processes are disabled. In such a state, although the guards that check some local invariants over the state of the process and its neighbors are disabled, in order to detect any perturbation of the global state, a silent stabilizing system continuously reevaluates certain local invariants to discover whether or not the invariants hold. This leads to the "pseudotermination" of stabilizing systems, where, although the global invariant has been satisfied and the state of the system has stabilized, the message transmission and/or the verification of the local invariants continues, causing the waste of the channel bandwidth, battery power, and other resources in the absence of demand for these services. A deactivation/reactivation protocol distinguishes itself from a silent stabilizing protocol in the following way: In a silent stabilizing system, upon system termination, the message exchange among system processes continues, whereas, in a deactivation/reactivation system, after reaching a state where no active process remains, the message exchange among system processes is suspended.

A related problem, *the wake-up problem*, is a well-studied problem in distributed computing [5]. Starting in a system state where all processes are asleep, a distributed wake-up protocol activates all the system processes after one or more of the system processes spontaneously wake up. In [6], Gasieniec et al. propose a wake-up protocol for synchronous broadcast systems. Fischer et al. in [7] present a solution to a variation of the wake-up problem resembling the consensus problem. They present a t-resilient protocol for n asynchronous processes in a shared-memory environment such that at least p processes eventually learn that at least τ processes have awakened and have begun participating in the protocol. In addition, protocols that determine the set of processes to set to sleep exist in the literature [8], [9], [3]. The stabilizing deactivation/reactivation protocol distinguishes itself from most existing wake-up and deactivation (sleeping) protocols in three ways. First, the stabilizing deactivation/reactivation protocol combines the deactivation and reactivation protocols. Second, the protocol achieves deactivation and reactivation of the system processes in bounded time. Third, these upper bounds are obtained starting from any arbitrary system configuration or after a transient fault in the system.

In this paper, we propose a novel synchronous self-stabilizing protocol for the deactivation of the distributed system services in the absence of demand and the reactivation of these services on demand. The proposed protocol is referred to as *the stabilizing deactivation/reactivation protocol* and has many applications in sensor, mobile, and ad hoc networks. Only a single-type message carrying no information is sent in the system and the message buffer for receiving these messages in each process can store at most one message.

The paper is organized as follows: Section 2 defines the properties of the deactivation/reactivation protocol. Section 3 contains an in-depth discussion of the system model, namely, the communication primitives used in the algorithm, the input, the variables, and the actions of the deactivation/reactivation system. Section 4 presents our self-stabilizing deactivation/reactivation protocol. In Sections 5 and 6, we provide a correctness proof of the deactivation/reactivation and determine the values of system parameters. We conclude the paper in Section 7 with some final remarks.

2 Properties of Deactivation/Reactivation Protocols

A deactivation/reactivation protocol is one that can be used by the processes in a distributed protocol to "deactivate" and "reactivate" all of the processes in the protocol. The main idea of such a protocol is as follows: When a process $p[i]$ needs to reactivate all of the processes in the protocol, process $p[i]$ becomes active and starts to broadcast "tick" messages periodically to its neighboring processes. When a neighboring process receives a $tick$ message, the neighboring process becomes active or stays active and broadcasts a $tick$ message to its neighboring processes. Then, the cycle repeats. Eventually, all of the processes in the protocol start to receive $tick$ messages periodically and stay active. Later, when process $p[i]$ no longer needs to keep all of the protocol processes active, $p[i]$ becomes inactive and stops broadcasting the $tick$ messages. When a process in the protocol does not receive any $tick$ message for some specified time period, the process becomes inactive. Thus, eventually, all of the processes in the protocol become inactive.

Our deactivation/reactivation protocol is also required to satisfy two real-time properties. First, there is a time period T_d such that, if no process needs to activate the protocol processes at any time during the interval $[t, t+T]$, where $T \geq T_d$, then all of the protocol processes are inactive during the time interval $[t+T_d, t+T]$. Second, there is a time period T_r such that, if one or more processes need to activate the protocol processes at any time during the interval $[t, t+T]$, where $T \geq T_r$, then all of the protocol processes are active during the time interval $[t+T_r, t+T]$. The time period T_d is called the *time period to deactivation* and the time period T_r is called the *time period to reactivation*.

In order that our deactivation/reactivation protocol will satisfy these real-time properties, the processes in the protocol need to execute synchronously (rather than asynchronously) and the message propagation delay between neighboring processes needs to be bounded from above. Next, we give an overview of the processes and their synchronous execution in our protocol.

Each process in our protocol has a Boolean input named "wakeup" and a Boolean variable named "active." The $wakeup$ input in each process $p[i]$ acts as the interface between the application and our protocol in $p[i]$. For instance, when the application in $p[i]$ needs to activate all of the processes in the protocol, the application assigns the value **true** to the $wakeup$ input in $p[i]$. Then, within T_r time units, each $active$ variable in each process in the protocol is assigned the value **true**. Later, when the application in $p[i]$ no longer needs to keep all the processes active, the application assigns the value **false** to the $wakeup$ input in $p[i]$. One of two outcomes is possible in this case. Within T_d time units, either the $active$ variable in $p[i]$ continues to have the value **true** (indicating that the application in another process still needs to keep all of the processes in the protocol active) or the $active$ variable in $p[i]$ is assigned the value **false** (indicating that no application in any process needs to keep all of the processes in the protocol active).

Each process in our protocol has two actions. Each action can read but not write the $wakeup$ input in its process and it can both read and write each variable in its process.

We assume that the real time is defined by an infinite sequence of discrete instants: instant 0, instant 1, instant 2, and so on. The time period between any two consecutive instants t and $t+1$, also called a time unit, is denoted by the time interval $(t, t+1)$. We assume that the enabled actions in all processes in the protocol are executed only at the discrete instants of real time and not during the time intervals between instants. (This implies that action execution takes zero time.)

The state of a process in our protocol during any time interval $(t, t+1)$ is defined by the value of each variable in this process during the time interval $(t, t+1)$. Note that, since no action is executed during any time interval, the value of every variable in every process and the state of every process in the protocol remain fixed during every time interval.

The state of our protocol during any time interval $(t, t+1)$ is defined by the collection of the states of all processes in the protocol during the time interval $(t, t+1)$. Note that the state of the protocol remains fixed during any time interval.

Let S_0 denote the state of our protocol during the time interval (0, 1). (Recall that this state remains fixed during the time interval (0, 1).) Then, at the next time instant, instant 1, every action (in every process in the protocol) that is enabled for execution is executed, yielding the next protocol state, denoted as S_1. State S_1 remains fixed during the time interval (1, 2). Then, at the next time instant, instant 2, every action (in every process in the protocol) is executed, yielding the next protocol state, denoted as S_2, and so on. The resulting infinite sequence of protocol states, $S_0, S_1, S_2, \cdots$, is called a computation of the protocol. We are now ready to formally specify the two real-time properties of our protocol:

1. *Deactivation*. In each computation $S_0, S_1, S_2, \cdots$ of the protocol where the $wakeup$ input in every process in the protocol has the value **false** at each protocol state, the $active$ variable in every process in the protocol has the value **false** at each of the protocol states $S_{T_d}, S_{T_d+1}, S_{T_d+2}, \cdots$.
2. *Reactivation*. In each computation $S_0, S_1, S_2, \cdots$ of the protocol where the $wakeup$ input in at least one process in the protocol has the value **true** at each protocol state, the $active$ variable in every process in the protocol has the value **true** at each of the protocol states $S_{T_r}, S_{T_r+1}, S_{T_r+2}, \cdots$.

Note that these two properties need to be satisfied for each protocol computation $S_0, S_1, S_2, \cdots$ whose initial protocol state S_0 is arbitrary. This indicates that the correctness of our protocol does not depend on the initial state of the protocol. Thus, our protocol is stabilizing.

3 Input, Variables, and Actions

In this section, we discuss the input, variables, and actions in each process in our deactivation/reactivation protocol.

The *topology* of the deactivation/reactivation protocol is an undirected graph where the following conditions hold: First, there is a one-to-one correspondence between the nodes in the graph and the processes in the protocol. Second, each edge between two nodes in the graph indicates that the two corresponding processes are "neighbors" in the protocol and, so, can send messages to one another.

Each process in the deactivation/reactivation protocol has one local input, four local variables, and a set of actions to be executed by the process. Next, we describe each of these six items in more detail.

The local input in each process is named $wakeup$. As mentioned in the previous section, a process can read but not write its $wakeup$ input and, so, the value of this input is determined by the application that uses this deactivation/reactivation protocol.

Each process has four local variables, named $active$, $accept$, *message buffer*, and $timer$. These variables are read and written only by the actions of their process.

The Boolean variable $active$ in a process indicates whether or not the process is active.

The Boolean variable $accept$ in a process is used by the process to limit the number of messages that the process forwards within the protocol. If the process receives a message when its $accept$ variable is **true**, then the process accepts the message, forwards a copy of the message to every neighboring process, and assigns **false** to the $accept$ variable for some specified time period. If the process receives a message when its $accept$ variable is **false**, then the process discards the message.

The *buffer* of each process holds at most one incoming message until that message is received by the process. (We assumed that these message buffers are used solely by the deactivation/reactivation protocol. Therefore, when this protocol is combined with other protocols, these other protocols need to use independent buffers.)

The processes in our protocol exchange one type of message, called a $tick$ message. For a process to send a $tick$ message, this process executes

broadcast $tick$.

Execution of this statement by a process causes one $tick$ message to arrive, after some propagation delay, at the message buffer of each neighboring process of the sending process. If a $tick$ message arrives at the buffer of a neighboring process when this buffer has a $(tick)$ message, then the newly arrived tick message is lost. Otherwise, the arrived $tick$ message is stored in the empty buffer. Later,

this $tick$ message is removed from the buffer of the neighboring process when this process executes

$$\textbf{rcv}\ tick.$$

For simplicity, we assume that the propagation delay of a $tick$ message is more than $zero$ and less than one time unit. Thus, if a process executes broadcast $tick$ at instant t, then the **broadcast** $tick$ message arrives at the buffer of a neighboring process during time interval $(t, t+1)$ and the neighboring process receives the $tick$ message at instant $t+1$.

Each process in the deactivation/reactivation protocol updates its $timer$ variable as follows: When a process executes a statement of the form

$$\textbf{timeout after} < period >,$$

the value of $timer$ in that process is set to $< period >$. At each time instant, the value of each $timer$ in each process in the protocol is synchronously decremented by one. When the value of $timer$ in a process $p[i]$ is decremented by one and becomes 0 at instant t, then $p[i]$ executes its **timeout** action at instant t. Each execution of a **timeout** action includes an execution of a statement of the form **timeout after** $< period >$. Because we assume that the execution of an action takes $zero$ time, the value of $timer$ in process $p[i]$ ends up being the value of $< period >$ at instant t.

Each action in each process in the deactivation/reactivation protocol is of the form

$$< guard > \rightarrow < sequence\ of\ statements >.$$

The $< sequence\ of\ statements >$ is executed only at an instant when the $< guard >$ of that action is **true**. Each process has two actions of the form

$$\textbf{rcv}\ tick \rightarrow < sequence\ of\ statements >,$$

$$\textbf{timeout} \rightarrow < sequence\ of\ statements >.$$

The first action is executed at instant t if the message buffer of the process has a $tick$ message at t. The second action is executed at instant t if the $timer$ of the process has value 0 at t.

We assume that the execution of an action takes $zero$ time. Thus, executing the two actions in a process, provided that their guards are **true** at t, starts at t and also finishes at t.

At each instant t, the actions of every process in the protocol are executed at instant t in three steps. First, if the *message buffer* of the process has a $tick$ message, then this message is removed from the *buffer* and the sequence of statements in the receiving action of the process is executed. Second, the value of the $timer$ variable of the process is decremented by one. Third, if the value of the $timer$ variable of the process is 0, then the sequence of statements in the **timeout** action of the process is executed.

4 The Deactivation/Reactivation Protocol

We are now ready to present our deactivation/reactivation protocol. The basic idea of this protocol is simple enough. When the $wakeup$ input of a process $p[i]$ is **true**, $p[i]$ makes its $active$ variable **true** and broadcasts a $tick$ message to all of its neighbors every Y time units. The broadcast $tick$ messages end up at the message buffer of every neighboring process of $p[i]$. When a process $p[j]$ receives a $tick$ message from its message buffer and it is ready to accept it, that is, its $accept$ variable is **true**, $p[j]$ makes its $active$ variable **true** and broadcasts a $tick$ message to all of its neighbors and the cycle repeats. When a process $p[i]$ detects that it has not received any $tick$ messages for a long time period, say, Z time units, process $p[i]$ concludes that the $wakeup$ input in every process in the protocol is **false**. In this case, $p[i]$ makes its own $active$ variable **false**.

Note that this protocol has two nice features. First, the protocol processes exchange only one type of message, namely, the $tick$ message. Second, the exchanged $tick$ messages have no fields.

Our deactivation/reactivation protocol, as described so far, has a problem. If a process $p[i]$ broadcasts even one $tick$ message (possibly because its $wakeup$ input has become **false** after broadcasting the first tick message and before broadcasting the second $tick$ message), then this one message can, over time, generate an unbounded number of $tick$ messages that populate the protocol indefinitely.

This problem can be solved as follows: Each process is provided with a Boolean local variable named $accept$. If a process $p[i]$ receives a $tick$ message when its $accept$ variable is **true**, $p[i]$ handles this message as above, that is, $p[i]$ makes its $active$ variable **true** and broadcasts $tick$ messages to every one of its neighbors, but makes its $accept$ variable **false** for a period of Y time units. If $p[i]$ receives a $tick$ message when its $accept$ variable is **false**, $p[i]$ discards the message.

The program for a process $p[i]$ in our deactivation/reactivation protocol is specified in Fig. 1. Note that the program of process $p[i]$ has two actions. The first action is executed only when the message buffer of $p[i]$ has a $tick$ message, and the second action is executed only when the $timer$ variable of $p[i]$ has the value $zero$.

Note that the value of $timer$ in each process $p[i]$ is updated as follows: First, the value of $timer$ becomes Y or Z when the statement **timeout after** Y or **timeout after** Z (respectively) is executed. Second, the value of $timer$ in each process $p[i]$ is decremented by one after each time unit. Because $Y < Z$, the range of values of $timer$ in each process $p[i]$ is $0 \ldots Z$.

The above specification of our deactivation/reactivation protocol is not complete. To complete this specification, we need to define four time periods: Y, Z, the time period to deactivation T_d, and the time period to reactivation T_r. In the next section, we choose a reasonable value for Y and use this value to compute T_d as a function of Y and Z. Then, in Section 6, we choose a reasonable value for Z and use this value to compute T_r as a function of Y and Z.

5 Analysis of Deactivation

Assume that the value of Y in our protocol satisfies the following condition:

$$Y = N, \tag{1}$$

where N is the number of processes in the protocol.

In this section, we show that, under (1), the above protocol satisfies the deactivation property discussed in Section 2. In

```
process p[i : 0 ··· n − 1]
const   Y, Z: integer                          {Y ≤ Z}
inp     wakeup: boolean
var     active : boolean
        accept : boolean

begin

        rcv tick  ⟶  if ¬accept                ⟶ skip;
                     ▯ accept                  ⟶ active:=true;
                                                 accept:=false;
                                                 broadcast tick;
                                                 timeout after Y
                     fi

        timeout   ⟶  if wakeup                 ⟶ active:=true;
                                                 accept:=false;
                                                 broadcast tick ;
                                                 timeout after Y
                     ▯ ¬wakeup ∧ ¬accept       ⟶ active:=true;
                                                 accept:=true;
                                                 timeout after Z
                     ▯ ¬wakeup ∧ accept        ⟶ active:=false;
                                                 timeout after Z
                     fi

end
```

Fig. 1. The stabilizing deactivation/reactivation protocol.

order to facilitate the presentation of Theorems 1 and 2 (below), we need the following definition.

Let M be a $tick$ message received and accepted by a process i at some instant. We know that, upon receipt of message M, process i may broadcast a $tick$ message M^1. In this case, message M^1 is said to be an *offspring* of message M, that is, $M^1 = offspring(M)$. This definition can be recursively generalized as follows: A message M' is said to be an offspring of message M iff there exists a sequence of messages $M^1, M^2, \cdots, M^k$ for $k > 1$ such that the following conditions hold: $M^1 = M$, for every $1 < i < k$, $M^i = offspring(M^{i-1})$, and $M^k = M'$.

In other words, M' is said to be an offspring of M iff M directly or indirectly causes M' through a number of $tick$ messages.

Theorem 1. *If a process in the deactivation/reactivation protocol receives and accepts a message and later receives any offspring of this message, the process is guaranteed to discard the offspring.*

Proof. If process P_i receives an offspring of message M that P_i sent at time t, then a cycle C of processes $P_i, P_1, P_2, \cdots, P_k, P_i$ exists such that P_1 receives and accepts message M sent by P_i at time t, for each process l, where $0 < l < k$, P_{l+1} receives and accepts an offspring of message M sent by P_l, and P_i receives an offspring of message M sent by P_k. Without loss of generality, let cycle C be of length at most N. Since the length of the cycle is at most N and each message transmission takes at most one time unit, an offspring of M arrives at P_i in the time interval $[t, t+N]$. Since process P_i assigns **false** to its $accept$ variable at time t and the $accept$ value remains **false** until the **rcv** action is executed at time $t + Y = t + N$ (see the algorithm), the offspring of M is discarded by process P_i. Hence, the theorem follows. □

Theorem 2. *If a process times out and broadcasts a tick message at a state S_t, then no offspring of this tick message is in some message buffer in the protocol at state S_{t+Y}.*

Proof. Let $P_0, P_1, P_2, \cdots P_k$ be a maximal sequence of processes such that P_1 receives and accepts message M sent by P_0 at time t, P_2 receives and accepts an offspring of message M sent by P_1 at time $t+1$, P_3 receives and accepts an offspring of message M sent by P_2 at time $t+2$, and so on.

By Theorem 1, we know that this sequence cannot contain a cycle. Therefore, the sequence does not have any repetitions. Since the number of processes is bounded by N, $k < N$ holds. Then, an offspring of M can be received but not accepted by a process after time $t + N - 1$. Consecutively, if an offspring of M is received at time $t + N$ by a process, it is removed from the message buffer and discarded and no offspring of M is sent by this process at time $t + N$. Hence, the theorem follows. □

Theorem 3 (Deactivation Theorem). *Consider a computation $S_0, S_1, \cdots$ of the deactivation/reactivation protocol and assume that the* wakeup *input of each process $p[i]$ in the protocol is* **false** *at every state S_i in this computation. At each of the states $S_{Y+2Z}, S_{Y+2Z+1}, \cdots$ in the computation, the* active *variable in every process in the protocol has the value* **false**.

Proof. First, observe that, since the $wakeup$ input is **false** for all processes at states $S_0, S_1, \cdots$, no **timeout** can cause a **broadcast** statement to be executed (see the algorithm). Then, at states $S_0, S_1, \cdots$ in the computation, each process $p[i]$ in the protocol broadcasts a $tick$ message only when process $p[i]$ receives a $tick$ message from its buffer when its $accept$ variable is **true**. This received message appears in the buffer either due to arbitrary initialization or due to the fact that $p[i]$ has received a $tick$ message in the computation $S_0, S_1, \cdots$. At state S_0, any $tick$ message of some message buffer is due to arbitrary initialization. Then, at states $S_1, S_2, \cdots$, any $tick$ message in some message buffer is an offspring of an initialization message. Now, by Theorem 2, we have that, at each of the states $S_Y, S_{Y+1}, \cdots$ in the computation, all message buffers are empty.

At state S_Y, the $accept$ variable of each process is either **true** or **false**. We now consider these two cases. In each case, we show that the $active$ variable of the process becomes **false**.

Case 1. The $accept$ variable for a process $p[i]$ is **true** at state S_Y.

We know that the $timer$ variable of process $p[i]$ contains at most Z at state S_Y. Now, notice that, in such a state, no guard other than the last branch of the **timeout** guard can ever be enabled. Then, it is easy to see that a **timeout** will be issued and the $active$ variable of the process becomes **false** at the latest at state S_{Z+Y}.

Case 2. The $accept$ variable for process $p[i]$ is **false** at state S_Y.

We know that the $timer$ variable of process $p[i]$ contains at most Z at state S_Y. Then, clearly, a **timeout** is

issued at the latest at state S_{Z+Y} and the *accept* variable of process $p[i]$ is assigned **true**. In addition, the *timer* variable of the process is assigned Z. Notice that, in such a state, no guard other than the last branch of the **timeout** guard can ever be enabled. Now, observe that a **timeout** will be issued and the *active* variable of the process becomes **false** at the latest at state S_{Y+2Z}.

It is easy to see that, after the *active* and *accept* variables of the process become **false** and **true**, respectively, at the latest at state S_{Y+2Z}, since no other action other than the one corresponding to the last branch of the **timeout** guard can ever be executed by process $p[i]$, the process continues to have **false** and **true** in its *active* and *accept* variables at states $S_{Y+2Z}, S_{Y+2Z+1}, \cdots$. Hence, the theorem follows. □

From Theorem 3, the deactivation period T_d of our protocol is given as follows:

$$T_d = Y + 2Z. \tag{2}$$

6 Analysis of Reactivation

Assume that the value of Z in our protocol satisfies the following condition:

$$Z = DY, \tag{3}$$

where D is the diameter of the protocol and Y satisfies (1).

In this section, we show that, under (3), our protocol satisfies the stabilization and reactivation properties discussed in Section 2. In order to facilitate the presentation of Theorems 4, 5, and 6, we need the following definition.

Let $S_0, S_1, \cdots,$ be a computation of the deactivation/reactivation protocol. State S_t of this computation is *proper* iff every process $p[i]$ in the protocol satisfies the following condition at S_t:

if *accept* in $p[i]$ is **false** at S_t
then *active* in $p[i]$ is **true** at S_t,
timer of $p[i]$ has a value $\leq Y$ at S_t, and
$p[i]$ has broadcasted a *tick* message at state $S_{t-(Y-k)}$
where k is the value of the *timer* at process $p[i]$ at state S_t

Theorem 4 (Stabilization Property). *In any computation $S_0, S_1, \cdots, S_Z, S_{Z+1}, \cdots$, each of the states $S_Z, S_{Z+1}, \cdots$ is proper.*

Proof. The initial value of the *timer* of any $p[i]$ is at most Z. Since the *timer* of each process $p[i]$ is decremented by one at each time instant, the *timer* of each process $p[i]$ becomes *zero* and its **timeout** guard becomes **true** no later than state S_Z. Then, when the **timeout** action of $p[i]$ is executed, one of the three branches of the if statement in this **timeout** action is executed before or at time Z. Observe that, each time one of the branches of this if statement is executed, one of the following holds:

Case 1. *wakeup* is **true**.

In this case, the *accept* variable of $p[i]$ is assigned **false**, the *active* variable of $p[i]$ is assigned **true**, the *timer* variable of $p[i]$ is assigned Y, and $p[i]$ broadcasts a *tick* message. Then, process $p[i]$ satisfies the proper condition.

Case 2. $\neg wakeup \wedge \neg accept$ is **true**.

In this case, the *accept* variable of process $p[i]$ is assigned **true** and, obviously, the proper condition holds for $p[i]$.

Case 3. $\neg wakeup \wedge accept$ is **true**.

In this case, the *accept* variable of process $p[i]$ is **true** initially and remains **true** after the execution of the action; hence, the proper condition holds for $p[i]$.

We showed that, after the **timeout** action of any process $p[i]$ is executed, the proper condition holds for each process at state S_Z or earlier. Observe that, whenever process $p[i]$ assigns **false** to its *accept* variable, it also assigns **true** to its *active* variable, broadcasts *tick* messages, and sets its *timer* variable to Y (see the second branch of the **rcv** guard and the first branch of the **broadcast** guard of the algorithm). Therefore, by the definition of the proper condition, we conclude that, after the proper condition holds for $p[i]$, the proper condition continues to hold for $p[i]$. Since the proper condition holds for each process at state S_Z or earlier and continues to hold afterward, each of the states $S_Z, S_{Z+1}, \cdots$ is proper. Hence, the theorem follows. □

When a *tick* message is received by a process, if the *accept* variable of the process is **false** due to arbitrary initialization, the received *tick* message is discarded, but no *tick* message is sent by the process prior to the receipt of *tick* message M. Otherwise, that is, if the *accept* variable is **false** when message M is received due to some action execution of the protocol, then a *tick* message must have been sent by the process prior to the receipt of M (see the actions of the protocol that assigns **false** to the *accept* variable). The last *tick* message, M', sent by the process prior to the receipt of message M is referred to as a *replacement message* of message M. In addition, any replacement message of M' is also referred to as a replacement message of message M and so on.

Theorem 5. *Consider a computation $S_0, S_1, \cdots$ of the deactivation/reactivation protocol and assume that the* wakeup *input of some process $p[i]$ in the protocol is* ***true*** *at every S_i in the computation. If process $p[i]$ sends* tick *message M at time t, where $t \geq Z + D$, process $p[j]$ other than process $p[i]$ may receive a replacement message of message M as early as time $(t + D) - (D - 1)Y$.*

Proof. Let $p[i], p[i_1], p[i_2], p[i_3], \cdots, p[i_k], p[j]$ be the shortest path of processes from process $p[i]$ to process $p[j]$ such that process $p[i]$ is a neighbor of process $p[i_1]$ that is a neighbor of process $p[i_2]$ and so on. A replacement message is sent along this path as follows: Message M sent by process $p[i]$ at time t is received by process $p[i_1]$ at time $t + 1$. If the *accept* variable of process $p[i_1]$ is **false** when message M arrives at process $p[i_1]$, process $p[i_1]$ may have sent a replacement *tick* message M^1 of M as early as time $(t + 1) - Y$ and assigned **false** to its *accept* variable. Analogously, when process $p[i_2]$ receives a *tick* message M^1 at time $(t + 2) - Y$ from process $p[i_1]$, if its *accept* variable is **false**, process $p[i_2]$ may have sent a replacement message M^2 of M^1 as early as time $(t + 2) - 2Y$ and so on. Recall that the replacement messages M^1 through M^{k+1} are replacement messages of message M.

Inductively, it can be shown that process $p[j]$ may receive the replacement $tick$ message M^k as early as time $(t + D) - (D - 1)Y$ from process $p[i_k]$. Hence, the theorem follows. □

Theorem 6. *Consider a computation $S_0, S_1, \cdots$ of the deactivation/reactivation protocol and assume that the* wakeup *input of some process $p[i]$ in the protocol is* ***true*** *at every S_i in the computation. If process $p[j]$ other than process $p[i]$ receives a tick message at state S_t, where $t \geq Z + D$, then $p[j]$ receives a tick message at state $S_{t'}$, where $t < t' \leq t + DY$.*

Proof. We first show that, if process $p[j]$ other than process $p[i]$ receives a $tick$ message at state S_t, where $t \geq Z + D$, then $p[j]$ receives a $tick$ message at state $S_{t'}$, where $t' > t$ and $t' - t \leq DY$. Consider two $tick$ messages, M_1 and M_2, broadcast by process $p[i]$ at times t and $t + Y$, where $t \geq Z + D$, respectively. In order to maximize the time gap between the receipts of two messages that are offsprings of messages M_1 and M_2 by an arbitrary process $p[j]$ in the protocol, we assume that message M_1 sent by process $p[i]$ is discarded and replaced by the oldest possible message in the protocol. On the other hand, message M_2 is not discarded and arrives at process $p[j]$ at its proper time with the largest possible delay.

By Theorem 5, we know that a message that is a replacement message of M_1 can be received as early as time $(t + D) - (D - 1)Y$. It is easy to see that if an offspring of message M_2 sent by process $p[i]$ at time $t + Y$ arrives at process $p[j]$ at the time with the largest delay, then this message arrives at process $p[j]$ at time $t + Y + D$. Therefore, the largest possible gap between the receipts of two messages is given by

$$t + Y + D((t + D) - (D - 1)Y),$$

which is equal to

$$DY.$$

Hence, the theorem follows. □

Theorem 7 (Reactivation Theorem). *Consider a computation $S_0, S_1, \cdots$, and assume that the* wakeup *input of some process $p[i]$ in the protocol is* **true** *at every S_i. At each of the states $S_{Z+2Y+D}, S_{Z+2Y+D+1}, \cdots$, the* active *variable in every process in the protocol has the value* **true**.

Proof. We first show that process $p[i]$ whose $wakeup$ input is **true** at each state of the computation has the value **true** in its $active$ variable at states $S_Z, S_{Z+1}, \cdots$. Since, at each time instant, the $timer$ in $p[i]$ is decremented by one, in a transition (S_k, S_{k+1}), where $k \leq Z - 1$, either the $timer$ of $p[i]$ becomes $zero$ and the **timeout** action of $p[i]$ is executed or process $p[i]$ receives a message, increments its $timer$, and sets its $active$ variable to **true**. In the latter case, we showed that the $active$ variable of process $p[i]$ becomes **true**. In the first case, that is, the first branch of the **timeout** action is executed, the $active$ variable of process $p[i]$ is assigned **true** at the latest at time $Z - 1$. It is easy to see that, for process $p[i]$ (and for each such process) whose $wakeup$ input remains **true**, once its $active$ variable becomes **true**, it remains **true**.

We now consider a process $p[j]$ whose $wakeup$ input remains **false** or fluctuates between **true** and **false** at states $S_0, S_1, \cdots$. Since process $p[i]$ has **true** in its $active$ variable and it broadcasts a $tick$ message in every time interval of size Y time units, we know that process $p[i]$ broadcasts a $tick$ message in time interval $[Z, Z + Y]$. By Theorem 5, we know that this $tick$ message can be replaced by a replacement $tick$ message that arrives at process $p[j]$ as early as state S_{2D+Y}. Observe that, if the message sent in this interval finds the $accept$ variables of the processes on all paths from process $p[i]$ to process $p[j]$ to be **true**, then an offspring of the message can arrive at process $p[j]$ at the latest at state S_{Z+Y+D}. Then, we can conclude that a message arrives at process $p[j]$ in time interval $[0, Z + Y + D]$.

Observe that, when process $p[j]$ receives a $tick$ message at the latest at state S_{Z+Y+D}, it finds the $accept$ variable of process $p[j]$ to be either **true** or **false**. If process $p[j]$ finds its $accept$ variable to be **true**, it is easy to see that it sets its $timer$ variable to Y and issues a **timeout** at the latest at state S_{Z+2Y+D}. Otherwise, that is, if it finds the $accept$ variable of process $p[j]$ to be **false**, since state S_{Z+Y+D} is proper, the $timer$ of process $p[j]$ contains a value at most Y and process $p[j]$ issues a **timeout** at state S_{Z+2Y+D}. Notice that, if process $p[j]$ (or any other process) receives a message followed by a **timeout** action execution, it sets its $active$ variable to **true**.

Now, we will show that, after the $active$ variable of process $p[j]$ is set to **true**, it remains **true**. Notice that, after process $p[j]$ executes a **timeout** action, either the $accept$ and the $timer$ variables of the process are set to **false** and Y, respectively, or the $accept$ and the $timer$ variables of the process are set to **true** and Z, respectively. Consider the first case, where the $accept$ and the $timer$ variables of the process are set to **false** and Y, respectively. Observe that the only way to set the $active$ variable of process $p[j]$ to **false** is to execute the third branch of the **timeout** guard. We know that, in order for the third branch of the **timeout** guard to be executed by process $p[j]$, its $accept$ variable should be equal to **true**. Then, process $p[j]$ has to execute the second branch of the **timeout** guard to set the $accept$ variable of process $p[j]$ to **true** prior to the execution of its third branch of the **timeout** guard. We know that, when the second branch of the **timeout** guard is executed, the implicit $timer$ variable of process $p[i]$ is set to Z. Then, observe that, in both of the aforementioned cases, prior to executing the third branch of the **timeout** guard, the $accept$ variable of process $p[j]$ is set to **true** and the $timer$ of the process is set to Z. If the third branch of the **timeout** guard is executed by process $p[j]$ after the process has Z in its $timer$ variable, then we know that process $p[j]$ does not receive a $tick$ message between the executions of these two branches for a period of Z time units. This is because, if a $tick$ message has been received between the executions of these two branches, the $accept$ variable of process $p[j]$ would have been set to **false** and the third branch would not have been executed. This contradicts Theorem 6. Hence, the theorem follows. □

From Theorem 6, it follows that the reactivation period T_r of our protocol is as follows:

$$T_r = 2Y + Z + D. \quad (4)$$

7 Concluding Remarks

We presented a novel self-stabilizing algorithm for the deactivation of distributed protocol services in the absence of demand and the reactivation of the services upon demand. In addition to being simple, self-stabilizing, and uniform, the following features of the algorithm make it desirable: First, single-size buffer per process, which can be implemented by a single bit, is used. Second, the messages of the algorithm contain no information. Third, no messages are sent when no process needs the service. Fourth, a limited number of messages are sent when some processes need the service.

The deactivation/reactivation protocol has two parameters: Y and Z. We showed Y to be at least N, where N is the number of processes in the deactivation/reactivation protocol. We also showed that Z needs to be greater than or equal to DY, where D is the network diameter in the deactivation/reactivation protocol. Consider a protocol where N is equal to 100 and D is about $logN$, which is about 7. For this network, Y needs to be at least 100 time units and Z needs to be at least 700 time units, where a time unit is about 100 milliseconds. This means that the process that needs the service needs to send a $tick$ message every 10 seconds. This also means that, when a process does not receive a $tick$ message for 2.5 minutes, it concludes that no process in the deactivation/reactivation protocol needs the service and deactivates itself.

One of the provisions of the proposed algorithm is the reduction in the number of messages exchanged in the presence of some active processes in the protocol to keep the protocol processes active. We now show an upper bound on the number of messages exchanged in the presence of some active processes. Consider a protocol state where k protocol processes are active and assume that all protocol processes remain active for T time units. First, observe that no protocol process broadcasts tick messages more than once during any time interval of size Y. Then, in the worst case, each channel can carry at most one message in each direction in every Y time units. Then, the number of messages sent in the protocol in T time units is $\lceil T/Y \rceil 2|E|$, where $|E|$ is the number of edges in the communication network.

We assumed that the propagation delay of messages is nonzero and at most one unit of time. It is straightforward to relax this assumption and assume an arbitrary but bounded message propagation delay. In addition, we assumed that processes execute their actions at each time instance in lock-step synchrony. It is an open problem to devise an asynchronous version of the proposed deactivation/reactivation protocol.

Acknowledgments

The authors would like to thank the anonymous referees for their suggestions and constructive comments on an earlier version of the paper. Their suggestions have greatly enhanced the readability of the paper.

References

[1] C.E. Perkins and E.M. Royer, "Ad-Hoc On-Demand Distance Vector Routing Protocol," *Proc. Second IEEE Workshop Mobile Computing Systems and Applications,* pp. 90-100, Feb. 1999.
[2] Y.-C. Hu, A. Perrig, and D.B. Johnson, "Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks," *Proc. MobiCom '02,* pp. 12-23, Sept. 2002.
[3] R. Kravets and R. Zheng, "On-Demand Power Management for Ad Hoc Networks," *Proc. INFOCOM '03,* Apr. 2003.
[4] S. Dolev, M.G. Gouda, and M. Schneider, "Memory Requirements for Silent Stabilization," *Acta Informatica,* vol. 36, no. 6, pp. 447-462, 1999.
[5] R. Zheng, J.C. Hou, and L. Sha, "Asynchronous Wakeup for Ad Hoc Networks," *Proc. MobiHoc '03,* pp. 35-45, 2003.
[6] L. Gasieniec, A. Pelc, and D. Peleg, "The Wakeup Problem in Synchronous Broadcast Systems (Extended Abstract)," *Proc. Symp. Principles of Distributed Computing,* pp. 113-121, 2000.
[7] M.J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld, "The Wakeup Problem," *SIAM J. Computing,* vol. 25, no. 6, pp. 1332-1357, Dec. 1996.
[8] L.M. Feeney and M. Nilsson, "Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment," *Proc. INFOCOM '01,* Apr. 2001.
[9] Y. Xu, J.S. Heidemann, and D. Estrin, "Geography-Informed Energy Conservation for Ad Hoc Routing," *Proc. MobiCom '01,* pp. 70-84, 2001.

Mehmet Hakan Karaata received the PhD degree in computer science in 1995 from the University of Iowa. He joined Bilkent University, Ankara, Turkey, as an assistant professor in 1995. He is currently working as an associate professor in the Department of Computer Engineering at Kuwait University. His research interests include mobile computing, distributed systems, fault-tolerant computing, and self-stabilization.

Mohamed G. Gouda received the BS degrees in engineering and mathematics from Cairo University in 1968 and 1971, respectively, the MA degree in mathematics from York University, Ontario, Canada, in 1972, and the MMath and PhD degrees in computer science from the University of Waterloo, Ontario, Canada, in 1973 and 1977, respectively. He currently holds the Mike A. Myers Centennial Professorship in Computer Sciences at the University of Texas at Austin. His research areas are distributed and concurrent computing and network protocols. In these areas, he has been working on abstraction, formality, correctness, nondeterminism, atomicity, reliability, security, convergence, and stabilization. He has published more than 15 book chapters, more than 60 journal papers, and more than 90 conference and workshop papers. He is the author of the textbook *Elements of Network Protocol Design* (John Wiley & Sons, 1998). This is the first ever textbook where network protocols are presented in an abstract and formal setting. He coauthored, with Tommy M. McGuire, the monograph *The Austin Protocol Compiler* (Springer, 2005). He also coauthored, with Chin-Tser Huang, the monograph *Hop Integrity in the Internet* (Springer, 2006). He is the 1993 winner of the Kuwait Award in Basic Sciences. He is also the recipient of an IBM Faculty Partnership Award for the academic year 2000-2001 and again for the academic year 2001-2002 and he became a fellow of the IBM Center for Advanced Studies in Austin in 2002. He won the 2001 IEEE Communication Society William R. Bennet Best Paper Award. In 2004, his paper "Diverse Firewall Design," coauthored with Alex X. Liu, won the William C. Carter Award. For more information, consult the Web site http://www.cs.utexas.edu/users/gouda.