Self-Stabilizing Philosophers with Generic Conflicts

PRAVEEN DANTURI, MIKHAIL NESTERENKO
Kent State University, USA
and
SÉBASTIEN TIXEUIL
Université Pierre & Marie Curie - Paris 6, France

We generalize the classic dining philosophers problem to separate the conflict and communication neighbors of each process. Communication neighbors may directly exchange information while conflict neighbors compete for the access to the exclusive critical section of code. This generalization is motivated by a number of practical problems in distributed systems including problems in wireless sensor networks. We present a self-stabilizing deterministic algorithm — $\mathcal{GDP}$ that solves this generalized problem. Our algorithm is terminating. We formally prove $\mathcal{GDP}$ correct and evaluate its performance. We extend the algorithm to handle a similarly generalized drinking philosophers and the committee coordination problem. We describe how $\mathcal{GDP}$ can be implemented in wireless sensor networks and demonstrate that this implementation does not jeopardize its correctness or termination properties.

Categories and Subject Descriptors: C.2.4 [**Computer-Communiation Networks**]: Distributed Systems; D.4.5 [**Operating Systems**]: Reliablity; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Algorithms, Reliability, Fault-Tolerance

Additional Key Words and Phrases: Self-Stabilization, Dining Philosophers

1. INTRODUCTION

Self-stabilization (or just stabilization) [Dolev 2000; Hoover 1995] is an elegant approach to forward recovery from transient faults as well as initializing a large-scale system. Regardless of the initial state, a stabilizing system converges to the legitimate set of states and remains there afterwards. In this paper we present a stabilizing solution to our generalization of the dining philosophers problem.

The dining philosophers problem [Dijkstra 1968; Chandy and Misra 1984; Lynch

A preliminary version of this article appeared in the proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06), July 2006, Dallas, TX.
M. Nesterenko was partially supported by NSF Career CNS-0347485. S. Tixeuil was partially supported by FNS grants FRAGILE and SR2I from ACI "Sécurité et Informatique", ANR Grants SOGEA and ALADDIN, and INRIA ARC FRACAS. S. Tixeuil did some of the the research for this paper while he was visiting Kent State University.

1980] is a fundamental resource allocation problem. The name of the problem is frequently shortened to *diners* [Chandy and Misra 1984; Sivilotti et al. 2000]. The diners, as well as its generalization — the drinking philosophers problem (drinkers) [Chandy and Misra 1984] and committee coordination problem [Chandy and Misra 1988], has a variety of applications. In diners, a set of processes (philosophers) request access to the critical section (CS) of code. For each process there is a set of neighbor processes. Each process has a conflict with its neighbors: it cannot share the CS with any of them. In spite of the conflicts, each requesting process should eventually execute the CS. To coordinate CS execution, the processes communicate. In classic diners it is assumed that each process can directly communicate with its conflict neighbors. In other words, for every process, the conflict neighbor set is a subset of the communication neighbor set.

However, there are applications where this assumption does not hold. Consider, for example, wireless sensor networks. A number of problems in this area, such as time division multiple access (TDMA) slot assignment, can be considered as resource allocation problems. Yet, due to radio propagation peculiarities, the signal's interference range may exceed its effective communication range. Moreover, radio networks have the so called hidden terminal effect. The problem is as follows. Let two transmitters t_1 and t_2 be mutually out of reception range and let receiver r be in range of them both. If t_1 and t_2 broadcast simultaneously, due to mutual radio interference, r is unable to receive either broadcast. The potential interference pattern is especially intricate if the antennas used by the wireless sensor nodes are directional (see for example [Malhotra et al. 2005]). Such transmitters can be modeled as conflict neighbors that are not communication neighbors. To accommodate such applications, we propose the following extension. Instead of one, each process has two sets of neighbors: the conflict neighbors and the communication neighbors. These two sets are not necessarily related. To make the problem solvable we require that each conflict-neighbor has to be reachable through the communication neighbors.

Some solutions to classic diners can potentially be extended to this problem. Indeed, if a separate communication channel is established to each conflict neighbor the classic diners program can be applied to the generalized case. However, such a solution may not be efficient. The channels to conflict neighbors go over the communication topology of the system. The channels to multiple neighbors of the same process may overlap. Moreover, the sparser the topology, the greater the potential overlap. Yet, in a diners program, the communication between conflict neighbors is only of two kinds: a process either requests the permission to execute the CS from the neighbors, or releases this permission. Due to channel overlap, separately communicating the same message to each conflict neighbor leads to excessive overhead. This motivates our search for a solution to generic diners that effectively combines communication to separate conflict neighbors.

Related work. There exist a number of deterministic self-stabilizing solutions to classic diners [G and Srimani 1999; Beauquier et al. 2000; Boulinier et al. 2004; Gouda and Haddix 1999; Huang 2000; Johnen et al. 2002; Mizuno and Nesterenko 1998; Nesterenko and Arora 2002b] and their weakenings [Gradinariu and Tixeuil

2007]. [Cantarell et al. 2003] solve the drinking philosophers problem. [Datta et al. 2005] solve a specific extension of the diners. None of these solutions separate conflict and communication neighbors.

Meanwhile, researchers working in the area of self-stabilization studied specific problems that require such separation. A few studies [Arumugam and Kulkarni 2005; Herman and Tixeuil 2004; Kulkarni and Arumugam 2003] address the aforementioned problem of TDMA slot assignment in the presence of the hidden terminal effect. This problem requires the processes to agree on a fixed schedule of time intervals (slots) such that each slot is allocated exclusively to a single process in the conflict neighborhood. Herman and Tixeuil [Herman and Tixeuil 2004] present a self-stabilizing probabilistic TDMA slot assignment algorithm for wireless sensor networks. They deal with channel conflicts that may arise between nodes that cannot communicate directly by assuming an underlying probabilistic carrier sense multiple access (CSMA) mechanism that provides expected constant time transmission. The authors assume that the network is tightly synchronized so that the phases that use the mechanism are clearly distinguished from the phases that use TDMA mechanism. [Arumugam and Kulkarni 2005; Kulkarni and Arumugam 2003] propose deterministic solutions to the same problem. In [Arumugam and Kulkarni 2005], to avoid conflicts they propose to serialize channel assignments by circulating a single assignment token (privilege) throughout the network. In [Kulkarni and Arumugam 2003], they consider a regular grid topology where each node is aware of its position in the grid. [Gairing et al. 2004] propose an elegant stabilizing algorithm for conflict neighbor sets containing the communication neighbors of distance at most two. They apply their algorithm to a number of graph-theoretical problems. However, their algorithm cannot solve the diners as it is not designed to ensure fair access to the CS in the case of several neighboring processes continually requesting it. That is, their program allow unfair computations. [Goddard et al. 2006] propose a solution to the conflict neighbor sets of communication neighbors at most k-hops away. Their solution recursively extends Gairing's algorithm. It is unfair as well.

Roadmap. We generalize the diners problem to separate the conflict and communication neighbor sets of each process. We formally state this problem, as well as describe our notation and execution model in Section 2. To the best of our knowledge, this problem has not been defined or addressed before either inside or outside the context of self-stabilization. In Section 3, we present $\mathcal{GDP}$ — a self-stabilizing deterministic terminating solution to this problem. In the same section we provide a formal correctness proof of $\mathcal{GDP}$ and discuss its performance. In Section 4 we describe how $\mathcal{GDP}$ can be implemented in wireless sensor networks. We describe a number of further extensions to $\mathcal{GDP}$ in Section 5. Specifically, we describe how $\mathcal{GDP}$ can be extended to solve the generalized drinkers problem and the generalized committee coordination problem; we simplify our solution to handle problems that do not require fairness of CS access.

2. PRELIMINARIES

Program model. For the formal description of our program we use simplified UNITY notation [Chandy and Misra 1988; Gouda 1998]. A program consists of a set of processes. A process contains a set of *constants* that it can read but not update. A process maintains a set of *variables*. Each variable ranges over a fixed domain of values. We use small case letters to denote singleton variables, and capital ones to denote sets. An action has the form $\langle name \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$. A *guard* is a Boolean predicate over the variables of the process and its communication neighbors. A *command* is a sequence of statements assigning new values to the variables of the process. We refer to a variable *var* and an action *ac* of process p as $var.p$ and $ac.p$ respectively. A *parameter* is used to define a set of actions as one parameterized action. For example, let j be a parameter ranging over values 2, 5, and 9; then a parameterized action $ac.j$ defines the set of actions: $ac.(j = 2) \;[\!]\; ac.(j = 5) \;[\!]\; ac.(j = 9)$.

A *state* of the program is the assignment of a value to every variable of each process from the variable's corresponding domain. Each process contains a set of actions. An action is *enabled* in some state if its guard is **true** at this state. A *computation* is a maximal fair sequence of states such that for each state s_i, the next state s_{i+1} is obtained by executing the command of an action that is enabled in s_i. Maximality of a computation means that the computation is infinite or it terminates in a state where none of the actions are enabled. Such state is a *fixpoint*. In a computation the action execution is *weakly fair*. That is, if an action is enabled in all but finitely many states of an infinite computation then this action is executed infinitely often.

A state *conforms* to a predicate if this predicate is **true** in this state; otherwise the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**. We do not differentiate between a set of states that conform to the predicate and the predicate itself. For example, when it is convenient we state that a state *belongs* to the predicate. Let R and S be predicates over the state of the program. Predicate R is *closed* with respect to the program actions if every state of the computation that starts in a state conforming to R also conforms to R. Predicate R *converges* to S if R and S are closed and any computation starting from a state conforming to R contains a state conforming to S. The program *stabilizes* to R if **true** converges to R.

Problem statement. An instance of the generic diners problem defines for each process p a set of *communication neighbors* $N.p$ and a set of *conflict neighbors* $M.p$. Both relations are symmetric. That is, for any two processes p and q if $p \in N.q$ then $q \in N.p$. The same applies to $M.p$. We do not consider asymmetric communication links that may, for example, appear in radio transmission. Throughout the computation each process requests CS access an arbitrary number of times: from zero to infinity. A program that solves the generic diners satisfies the following two properties for each process p:

. *safety* — if the action that executes the CS is enabled in p, it is disabled in all processes of $M.p$;

. *liveness* — if p wishes to execute the CS, then the computation contains either

```
process $p$
const
    $M$: conflict neighbors of $p$
    $N$: communication neighbors of $p$
    $(\forall q : q \in M : dad.p.q \in N, KIDS.p.q \subset N)$
            parent id and set of children ids for each conflict neighbor
parameter
    $r : M$
var
    $state.p.p : \{\mathbf{idle}, \mathbf{req}\}$,
    $(\forall q : q \in M : state.p.q : \{\mathbf{idle}, \mathbf{req}, \mathbf{rep}\})$,
    $YIELD : \{\forall q : q \in M : q > p\}$ lower priority processes to wait for
    $needcs$ : boolean, application variable to request the CS

                *[
join:               $needcs \wedge state.p.p = \mathbf{idle} \wedge YIELD = \varnothing \wedge$
                    $(\forall q : q \in KIDS.p.p : state.q.p = \mathbf{idle}) \longrightarrow$
                        $state.p.p := \mathbf{req}$
                 []
enter:              $state.p.p = \mathbf{req} \wedge$
                    $(\forall q : q \in KIDS.p.p : state.q.p = \mathbf{rep}) \wedge$
                    $(\forall q : q \in M \wedge q < p : state.p.q = \mathbf{idle}) \longrightarrow$
                        /* CS */
                        $YIELD := \{\forall q : q \in M \wedge q > p : state.p.q = \mathbf{rep}\}$,
                        $state.p.p := \mathbf{idle}$
                 []
forward.r:          $state.p.r = \mathbf{idle} \wedge state.(dad.p.r).r = \mathbf{req} \wedge$
                    $((KIDS.p.r = \varnothing) \vee (\forall q : q \in KIDS.p.r : state.q.r = \mathbf{idle})) \longrightarrow$
                        $state.p.r := \mathbf{req}$
                 []
back.r:             $(state.p.r = \mathbf{req} \wedge state.(dad.p.r).r = \mathbf{req} \wedge$
                    $((KIDS.p.r = \varnothing) \vee (\forall q : q \in KIDS.p.r : state.q.r = \mathbf{rep}))) \vee$
                    $(state.p.r \neq \mathbf{rep} \wedge state.(dad.p.r).r = \mathbf{rep}) \longrightarrow$
                        $state.p.r := \mathbf{rep}$
                 []
stop.r:             $(state.p.r \neq \mathbf{idle} \vee r \in YIELD) \wedge$
                    $state.(dad.p.r).r = \mathbf{idle} \longrightarrow$
                        $YIELD := YIELD \setminus \{r\}$,
                        $state.p.r := \mathbf{idle}$
                 ]
```

Fig. 1. Process of $\mathcal{GDP}$

the execution of the CS or a state where p does not wish to enter the CS.

A desirable performance property of a solution to diners is *termination*: if a computation contains finitely many states where processes wish to execute the CS, then this computation is itself finite. To put another way, if there are no requests for the CS, a terminating solution to diners should eventually arrive at a state where no actions are enabled.

3. $\mathcal{GDP}$ ALGORITHM

3.1 Description

Algorithm overview. The main idea of the algorithm is to coordinate CS request notifications between multiple conflict neighbors of the same process. We assume that for each process p there is a tree that spans the conflict set $M.p$. This tree is rooted in p. For simplicity, we assume that every process q that is in the tree also belongs to $M.p$. A stabilizing breadth-first construction of a spanning tree is a relatively simple task [Dolev 2000]. It is well-known that layered composition preserves the property of stabilization [Herman 1991]: if the two component programs are independently stabilizing and one of the components does not modify the behavior of the other, then the combined system is stabilizing as well. Essentially, after the tree-construction component stabilizes and outputs the correct tree, the diners may start to stabilize. This property allows us to ignore the spanning tree formation and assume that the requisite variables are available to our diners algorithm as constants.

The processes in the spanning tree propagate the CS request of its root. The request reflects from the leaves and informs the root that its conflict neighbors are notified. This mechanism resembles information propagation with feedback [Blin et al. 2003; Bui et al. 1999].

The access to the CS is granted on the basis of the priority of the requesting process. Each process has an identifier that is unique throughout the system. A process with lower identifier has higher priority. To ensure liveness, when executing the CS, each process p records the identifiers of its lower priority conflict neighbors that also request the CS. Before requesting it again, p then waits until all these processes access the CS. This technique resembles a synchronization mechanism in the atomicity refinement solution by Nesterenko and Arora [Nesterenko and Arora 2002b].

Detailed description. Each process p has access to a number of constants. The set of identifiers of its communication neighbors is N, and its conflict neighbors is M. For each of its conflict neighbors r, p knows the appropriate spanning tree information: the parent identifier — $dad.p.r$, and a set of ids of its children — $KIDS.p.r$.

Process p stores its own request state in variable $state.p.p$ and the state of each of its conflict neighbors in $state.p.r$. Notice that p's own state can be only **idle** or **req**, while for its conflict neighbors p also has **rep**. To simplify the description, depending on the state, we refer to the process as being idle, requesting or replying. In $YIELD$, process p maintains the ids of its lower priority conflict neighbors that should be allowed to enter the CS before p requests it again. Variable $needcs$ is an external Boolean variable that indicates if CS access is desired. Notice that CS entry is guaranteed only if $needcs$ remains **true** until p requests the CS.

There are five actions in the algorithm. The first two: *join* and *enter* manage CS entry of p itself. The remaining three: *forward*, *back* and *stop* — propagate CS request information along the tree. Notice that the latter three actions are parameterized over the set of p's conflict neighbors.

Action *join* states that p requests the CS when the application variable $needcs$

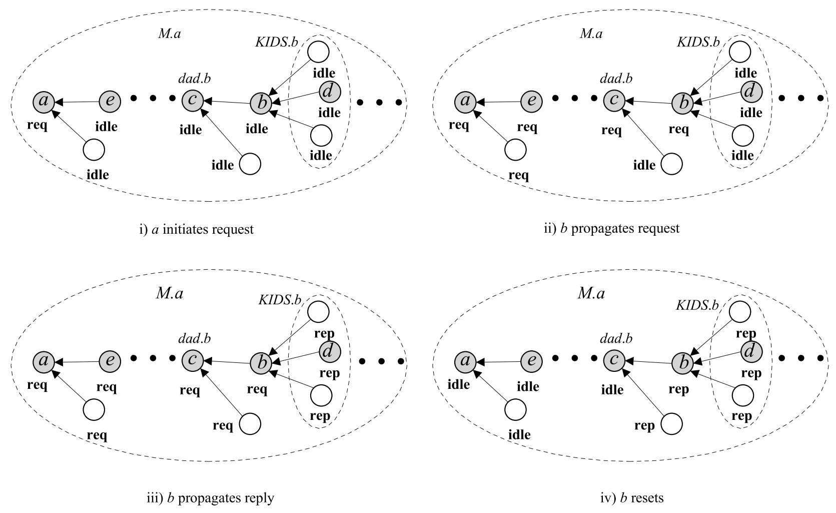

Fig. 2. Phases of $\mathcal{GDP}$ operation

is **true**, p itself, as well as its children in its own spanning tree, is idle and there are no lower priority conflict neighbors to wait for. As action *enter* describes, p enters the CS when its children reply and the higher priority processes do not request the CS themselves. To simplify the presentation, we describe the CS execution as a single action[1].

Action *forward* describes the propagation of a request of a conflict neighbor r of p along r's tree. Process p propagates the request when p's parent — $dad.p.r$ is requesting and p's children are idle. Similarly, *back* describes the propagation of a reply back to r. Process p propagates the reply either if its parent is requesting and p is the leaf in r's tree or all p's children are replying. The second disjunct of *back* is to expedite the stabilization of $\mathcal{GDP}$. Action *stop* resets the state of p in r's tree to idle when its parent is idle. This action removes r from the set of lower-priority processes to await before initiating another request.

Example operation. The operation of $\mathcal{GDP}$ in legitimate states is illustrated in Figure 2. We focus on the conflict neighborhood $M.a$ of a certain node a. We consider representative nodes in the spanning tree of $M.a$. Specifically, we consider one of a's children — e, a descendant — b, b's parent — c and one of b's children — d.

Initially, the states of all processes in $M.a$ are idle. Then, a executes *join* and sets $state.a.a$ to **req** (see Figure 2, i). This request propagates to process b, which executes *forward* and sets $state.b.a$ to **req** as well (Figure 2, ii). The request reaches the leaves and bounces back as the leaves change their state to **rep**. Process b then executes *back* and changes its state to **rep** as well (Figure 2, iii). After the reply

[1]Separating CS exit into a separate action is not difficult: as the process enters the CS, due to safety, its conflict neighbors are blocked from CS entry and have to wait for the progress of this process. Hence, CS exit can be relegated to a separate action.

reaches a and if none of the higher priority processes are requesting the CS, a executes *enter*. This action resets $state.a.a$ to **idle**. This reset propagates to b which executes *stop* and also changes $state.b.a$ to **idle** (Figure 2, iv).

3.2 Proof of Correctness

Proof outline. We present the$\mathcal{GDP}$ correctness proof as follows. We first state a predicate we call $InvG$ and demonstrate that $\mathcal{GDP}$ stabilizes to it in Theorem 1. We then proceed to show that if $InvG$ holds, then $\mathcal{GDP}$ satisfies the safety and liveness properties of the generic diners in Theorems 2 and 3 respectively.

Proof notation. Throughout this section, unless otherwise specified, we consider the conflict neighbors of a certain node a (see Figure 2). That is, we implicitly assume that a is universally quantified over all processes in the system. We focus on the following nodes: $e \in KIDS.a.a$, $b \in M.a$, $c \equiv dad.b.a$ and $d \in KIDS.b.a$.

Since we discuss the states of e, b, c and d in the spanning tree of a, when it is clear from the context, we omit the specifier of the conflict neighborhood. For example, we use $state.b$ for $state.b.a$. Also, for clarity, we attach the identifier of the process to the actions it contains. For example, *forward.b* is the *forward* action of process b.

Our global predicate consists of the following predicates that constrain the states of each individual process and the states of its communication neighbors. The predicate below relates the states of the root of the tree a to the states of its children.

$$(state.a = \textbf{idle}) \Rightarrow (\forall e : e \in KIDS.a : state.e \neq \textbf{req}) \qquad (Inv.a)$$

The following sequence of predicates relates the state of b to the state of its neighbors.

$$state.b = \textbf{idle} \wedge state.c \neq \textbf{rep} \wedge (\forall d : d \in KIDS.b : state.d \neq \textbf{req}) \qquad (I.b.a)$$

$$state.b = \textbf{req} \wedge state.c = \textbf{req} \qquad (R.b.a)$$

$$state.b = \textbf{rep} \wedge \qquad (\forall d : d \in KIDS.b : state.d = \textbf{rep}) \qquad (P.b.a)$$

We denote the disjunction of the above three predicates as follows:

$$I.b.a \vee R.b.a \vee P.b.a \qquad (Inv.b.a)$$

The following predicate relates the states of all processes in $M.a$.

$$(\forall a :: Inv.a \wedge (\forall b : b \in M.a : Inv.b.a)) \qquad (InvG)$$

To aid in exposition, we mapped the states and transitions for individual processes in Figure 3. Note that to simplify the picture, for the intermediate process b we only show the states and transitions if Inv holds for each ancestor of b. For b, the $I.b$, $R.b$ and $P.b$ denote the states conforming to the respective predicates. While the primed versions $I'.b$ and $P'.b$ signify the states where b is respectively idle and replying but $Inv.b.a$ does not hold. Notice that if $Inv.c$ holds for b's parent c, the primed version of R does not exist. Indeed, to violate R, b should be requesting while c is either idle or replying. However, if $Inv.c$ holds and c is in either of these two states, b cannot be requesting.

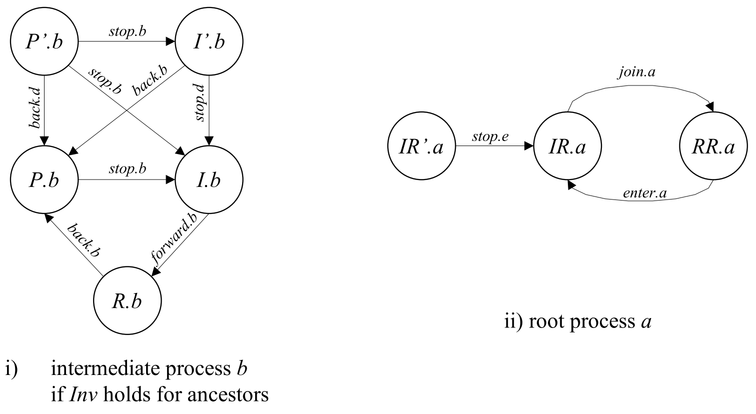

Fig. 3. State transitions for an individual process

For a, $IR.a$ and $RR.a$ denote the states where a is respectively idle and requesting while $Inv.a$ holds. In states $IR'.a$, a is idle while $Inv.a$ does not hold. Notice that since $state = \mathbf{req}$ falsifies the antecedent of $Inv.a$, the predicate always holds if a is requesting. The state transitions in Figure 3 are labeled by actions whose execution effects them. Loopback transitions are not shown.

THEOREM 1 (*Stabilization*). Program $\mathcal{GDP}$ stabilizes to $InvG$.

PROOF. By the definition of stabilization, $InvG$ should be closed with respect to the execution of the actions of $\mathcal{GDP}$, and $\mathcal{GDP}$ should converge to $InvG$. We prove the closure first.

Closure. To aid in the subsequent convergence proof, we show a property that is stronger than just the closure of $InvG$. We demonstrate the closure of the following conjunction of predicates: $Inv.a$ and $Inv.b.a$ for a set of descendants of a up to a certain depth of the tree. To put another way, in showing the closure of $Inv.b.a$ for b we assume that the appropriate predicates hold for all its ancestors. Naturally, the closure of $InvG$ follows. By definition of predicate closure, we need to demonstrate that if the predicate holds in a certain state, the execution of any action in this state does not violate the predicate.

Let us consider $Inv.a$ and the root process a first. Notice that the only two actions that can potentially violate $Inv.a$ are *enter.a* and *forward.e*. Let us examine each action individually. If *enter.a* is enabled, each child of a is replying. Hence, when it is executed and it changes the state of a to **idle**, $Inv.a$ holds. If *forward.e* is enabled, a is requesting. Thus, executing the action and setting the state of e to **req** does not violate $Inv.a$.

Let us now consider $Inv.b.a$ for an intermediate process $b \in M.a$. We examine the effect of the actions of b, b's parent — c, and one of b's children — d in this sequence.

We start with the actions of b. If $I.b$ holds, *forward.b* is the only action that can be enabled. If it is enabled, c is requesting. Thus, if it is executed, $R.b$ holds and $Inv.b.a$ is not violated. If $R.b$ holds then *back.b* is the only action that can be enabled. However, if *back.b* is enabled and $R.b$ holds, then all children of b are replying. If *back.b* is executed, the resultant state conforms to $P.b$. If $P.b$ holds, then *stop.b* can exclusively be enabled. If $P.b$ holds and *stop.b* is enabled, then c is idle and all children of b are replying. The execution of *back.b* sets the state of b to **idle**. The resulting state conforms to $I.b$ and $Inv.b.a$ is not violated.

Let us examine the actions of c. Recall that we are assuming that *Inv.c* and the respective invariants of all of b's ancestors hold. If $I.b$ holds, *forward.c* and *join.c* (in case b is a child of a) are the actions that can possibly be enabled. If either is enabled, b is idle. The execution of either action changes the state of c to **req**. $I.b$ and $Inv.b.a$ still hold. If $R.b$ holds, none of the actions of c are enabled. Indeed, actions *forward.c*, *back.c*, *join.c* and *enter.c* are disabled. Moreover, if $R.b$ holds, c is requesting: since *Inv.c* holds, c must be in $R.c$. Which means that c's parent is not idle. Hence, *stop.c* is also disabled. Since $P.b$ does not mention the state of c, the execution of c's actions does not affect the validity of $P.b$.

Assume b has at least one child d. Let us examine its actions. If $I.b$ holds, the only possibly enabled action is *stop.d*. The execution of this action changes the state of d to **idle**, which does not violate $I.b$. $R.b$ does not mention the state of d. Hence, its action execution does not affect $R.b$. If $P.b$ holds, all actions of d are disabled. This concludes the closure proof of $InvG$.

Convergence. We prove convergence by induction on the depth of the tree rooted in a. Let us show convergence of a. The only illegitimate set of states is $IR'.a$. When a conforms to $IR'.a$, a is idle and at least one child e is requesting. In such state, all actions of a that affect its state are disabled. Moreover, for every child of a that is idle, all relevant actions are disabled as well. For the child of a that is not idle, the only enabled action is *stop.e*. After this action is executed, e is idle. Thus, eventually $IR.a$ holds.

Let a conform to $Inv.a$. Also, let every descendant process f of a up to depth i confirm to $Inv.f$. Let the distance from a to b be $i + 1$. We shall show that $Inv.b.a$ eventually holds. Notice that according to the preceding closure proof, the conjunction of $Inv.a$ and $Inv.f$ for each process f in the distance no more than i is closed.

Note that according to Figure 3, there is no loop in the state transitions containing primed states. Hence, to prove that b eventually satisfies $Inv.b.a$ we need to show that b does not remain in a single primed set of states indefinitely. Process b can satisfy either $I'.b$ or $P'.b$. Let us examine these cases individually.

Let $b \in I'.b$. Since *Inv.c* holds, if b is idle, c cannot satisfy $P.c$. Thus, for b to satisfy $I'.b$, at least one child d of b must be requesting. However, if b is idle then *stop.d* is enabled. Notice that when b is idle, none of its non-requesting children can start to request. Thus, when this *stop* is executed for every requesting child of b, b leaves $I'.b$.

Suppose $b \in P'.b$. Process b may leave $P'.b$ be executing *stop.b*. Assume it does not. If $b \in P'.b$, then there exists at least one child d of b that is not replying. However, for every such process d, *back.d* is enabled. Notice that when b is replying, none of its replying children can change state. Thus, when *back* is executed for every non-replying child of b, b leaves $P'.b$.

Hence, $\mathcal{GDP}$ converges to *InvG*. □

THEOREM 2 (*Safety*). If *InvG* holds and *enter.a* is enabled, then for every process $b \in M.a$, *enter.b* is disabled.

PROOF. If *enter.a* is enabled, every child of a is replying. Due to *InvG*, this means that every descendant of a is also replying. However, for each process $x \in M.a$ whose priority is lower than a, *enter.x* can only be enabled when *state.x.a* is **idle**. Thus, for every such process x, if *enter.a* is enabled, *enter.x* is disabled. Note also, that since *enter.a* is enabled, for every process $y \in M.a$ whose priority is higher than a's, *state.a.y* is **idle**. According to *InvG*, none of the ancestors of a in y's tree, including y's children, are replying. Thus, *enter.y* is disabled. In short, when *enter.a* is enabled, neither higher nor lower priority processes of $M.a$ have *enter* enabled. The theorem follows. □

LEMMA 1. If *InvG* holds and some process a is requesting, then eventually either a stops requesting or none of its descendants are idle.

PROOF. Notice that the lemma trivially holds if a stops requesting. Thus, we focus on proving the second claim of the lemma. We prove it by induction on the depth of a's tree. Process a is requesting and so it is not idle. By the assumption of the lemma, a never becomes **idle**. Now let us assume that this lemma holds for all its descendants up to distance i. Let b be a descendant of a whose distance from a is $i + 1$. And let b be idle.

By inductive assumption, b's parent c is not idle. Due to *InvG*, if b is idle, c is not replying. Hence, c is requesting. If there exists a child d of b that is not idle, then *stop.d* is enabled at d. When *stop.d* is executed, d is idle. Notice that when b and d are idle, all actions of d are disabled. Thus, d remains **idle**. When all children of b are idle and its parent is requesting, *forward.b* is enabled. When it is executed, b is not idle. Notice, that the only way for b to become idle again is to execute *stop.b*. However, by inductive assumption c is not idle. This means that *stop.b* is disabled. The lemma follows. □

LEMMA 2. If *InvG* holds and some process a is requesting, then eventually all its children in $M.a$ are replying.

PROOF. Notice that when a is requesting, the conditions of Lemma 1 are satisfied. Thus, eventually, none of the descendants of a are idle. Notice that if a process is replying, it does not start requesting without being idle first (see Figure 3). It remains to be proven that each individual process is eventually replying. We prove it by induction on the height of a's tree.

If a leaf node b is requesting and its parent is not idle, *back.b* is enabled. When it is executed, b is replying. Assume that each node whose longest distance to a leaf of a's tree is i is replying. Let b's longest distance to a leaf be $i+1$. By assumption, all its children are replying. Due to Lemma 1, its parent is not idle. In this case *back.b* is enabled. After it is executed, b is replying. By induction, the lemma holds. □

LEMMA 3. If *InvG* holds and the computation contains infinitely many states where a is idle, then for every descendant there are infinitely many states where it is idle as well.

Proof. We first consider the case where the computation contains a suffix where a is idle in every state. In this case we prove the lemma by induction on the depth of a's tree with a itself as a base case. Assume that there is a suffix where all descendants of a up to depth i are idle. Let us consider process b whose distance to a is $i+1$ and this suffix. Notice that this means that c remains idle in every state of this suffix. If b is not idle, *stop.b* is enabled. Once it is executed, no relevant actions are enabled at b and it remains idle afterwards. By induction, the lemma holds.

Let us now consider the case where no computation suffix of continuously idle a exists. Yet, there are infinitely many states where a is idle. Thus, a leaves the idle state and returns to it infinitely often. We prove by induction on the depth of the tree that every descendant of a behaves similarly. Assume that this claim holds for the descendants up to depth i. Let b's distance to a be $i+1$.

When $InvG$ holds, the only way for b's parent c to leave **idle** is to execute *forward.c* (see Figure 3). Similarly, the only way for c to return to **idle** is to execute *stop.c* while c is replying [2]. However, *forward.c* is enabled only when b is **idle**. Also, according to $InvG$ when c is requesting, b is not **idle**. Thus, b leaves **idle** and returns to it infinitely many times as well. By induction, the lemma follows. □

Lemma 4. If $InvG$ holds and process a is requesting such that and a's priority is the highest among the processes of $M.a$ that request the CS in this computation, then a eventually executes the CS.

Proof. If a is requesting, then, by Lemma 2, all its children are eventually replying. Therefore, the first and second conjuncts of the guard of *enter.a* are **true**. If a's priority is the highest among all the requesting processes in $M.a$, then each process z, whose priority is higher than that of a is idle. According to Lemma 3, *state.a.z* is eventually **idle**. Thus, the third and last conjunct of *enter.a* is enabled. This allows a to execute the CS. □

Lemma 5. If $InvG$ holds and process a is requesting, a eventually executes the CS.

Proof. Notice that by Lemma 2, for every requesting process, the children are eventually replying. According to $InvG$, this implies that all the descendants of the requesting process are also replying. For the remainder of the proof we assume that this condition holds.

We prove this lemma by induction on the priority of the requesting processes. According to Lemma 4, the requesting process with the highest priority eventually executes the CS. Thus, if process a is requesting and there is no higher priority process $b \in M.a$ which is also requesting then, by Lemma 4, a eventually enters the CS.

Suppose, on the contrary, that there exists a requesting process $b \in M.a$ whose priority is higher than a's. If every such process b enters the CS finitely many times, then, by repeated application of Lemma 4, there is a suffix of the computation where all processes with priority higher than a's are idle. Then, by Lemma 4, a enters the

[2]The argument is slightly different for $c = a$ as it executes *join.a* and *enter.a* instead.

CS. Suppose there exists a higher priority process b that enters the CS infinitely often. Since a is requesting, eventually $\mathit{state.b.a} = \mathbf{rep}$. When b executes the CS, it enters a into $\mathit{YIELD.b}$. We assume that b enters the CS infinitely often. However, b can request the CS again only if $\mathit{YIELD.b}$ is empty. The only action that takes a out of $\mathit{YIELD.b}$ is $\mathit{stop.b}$. However, this action is enabled if $\mathit{state.b.a}$ is **idle**. Notice that, if InvG holds, the only way for the descendants of a to move from replying to idle is if a itself moves from requesting to idle. That is a executes the CS. Thus, each process a requesting the CS eventually executes it. □

Lemma 6. If InvG holds and process a wishes to enter the CS, a eventually requests.

Proof. We show that a wishing to enter the CS eventually executes $\mathit{join.a}$. We assume that a is idle and $\mathit{needcs.a}$ is **true**. Then, $\mathit{join.a}$ is enabled if $\mathit{YIELD.a}$ is empty. Note that a adds a process to YIELD only when it executes the CS. Thus, as a remains idle, processes can only be removed from $\mathit{YIELD.a}$.

Let us consider a process $b \in \mathit{YIELD.a}$. If b executes the CS finitely many times, then there is a suffix of the computation where b is idle. According to Lemma 3, for all descendants of b, including a, $\mathit{state.a.b}$ is idle. If this is the case $\mathit{stop.a}$ is enabled. When it is executed b is removed from $\mathit{YIELD.a}$.

Let us consider the case, where b executes the CS infinitely often. In this case, b enters and leaves **idle** infinitely often. According to Lemma 3, $\mathit{state.a.b}$ is idle infinitely often. Moreover, a moves to idle by executing $\mathit{stop.a}$, which removes b from $\mathit{YIELD.a}$. The lemma follows. □

The theorem below follows from Lemmas 5 and 6.

Theorem 3 (*Liveness*). If InvG holds, a process wishing to enter the CS then the computation contains either the execution of the CS or a state where p does not wish to enter the CS.

We draw the following corollary from Theorems 1, 2 and 3.

Corollary 1. Program $\mathcal{GDP}$ is a self-stabilizing solution to the generic diners problem.

Theorem 4 (*Termination*). Program $\mathcal{GDP}$ is terminating.

Proof. To prove termination we need to show that if a computation of $\mathcal{GDP}$ contains only finitely many states where some process p is requesting (i.e. variable $\mathit{needcs.p}$ is **true**), then the computation ends in a fixpoint (a state where all actions are disabled). If such a computation ends in a state where $\mathit{needcs.p}$ is **true** for some process, then the condition of the theorem is satisfied since such computation is finite. Otherwise, the computation contains a suffix where $\mathit{needcs.p}$ is **false** for every process. Let us consider such computation σ and a single process a. According to Theorem 1, every computation contains a suffix where the invariant InvG holds. Let us consider the shorter of the two suffixes σ'. If the suffixes are of the same length (i.e. they are identical), we just let σ' be that suffix.

According to Theorem 3, if InvG holds and a is requesting to enter the CS in σ', a eventually does so. After CS entry, a becomes idle. Notice that a may not rejoin the CS contention in σ' as $\mathit{needcs.a}$ is **false**. Hence, a spends the remainder of σ'

as idle. A simple induction on the depth of the spanning tree in $M.a$ (cf. the first part of the proof of Lemma 3) demonstrates that there is a suffix σ'' of σ' where all descendants of a are also idle.

Let us consider the actions of a and its descendants in σ''. Observe that all of the processes are idle. Thus, the actions *enter*, *forward* and *back* are disabled. Action *join.a* is disabled due to $needcs.a = \textbf{idle}$. However, *stop.b* at some process b may be enabled if *YIELD.b* contains a. However, after *stop.b* executes, a is removed from *YIELD.b*. Process a is never included into *YIELD.b* again since *state.b.a* is idle in σ''. Hence, all actions of processes in $M.a$ are eventually disabled. Since this argument applies to all processes in the system, the computation σ ends in a fixpoint. The theorem follows. □

3.3 Efficiency Evaluation

We estimate the efficiency of our algorithm in asynchronous rounds. A *round* is a segment of a computation where each process that has an action enabled in the initial state, either executes this action or the action is disabled.

Let d and δ be the maximum depth of a conflict tree and the maximum degree of a process respectively. Observe (see Figure 3) that each process executes at most two of its own actions before satisfying the stabilization predicate. Thus, a conflict neighborhood stabilizes in $2(\delta+1)d$ rounds. The stabilization of one conflict neighborhood is independent of stabilization of another.

Synchronization delay is the maximum number of rounds between one conflict neighbor leaving the CS and another process joining it. The process CS entry and exit requires propagation of a request down the tree, collecting a reply and resetting the tree to its idle state. Thus, the synchronization delay of $\mathcal{GDP}$ is $3d$.

Step complexity is the maximum number of steps required per CS execution. Each process in the conflict tree has to execute exactly three steps. There are at most δ^d processes in the conflict tree. Therefore, the step complexity of the CS access of $\mathcal{GDP}$ is $3\delta^d$.

4. IMPLEMENTATION IN WIRELESS SENSOR NETWORKS

As we motivated $\mathcal{GDP}$ by the problems arising in wireless sensor networks, we would like to discuss the implementation of our algorithm in this environment. A spanning tree construction algorithm for such environment is available [Nesterenko and Arora 2001]. Let us attend to the stabilization of diners proper.

From algorithm correctness standpoint, this environment is a variant of a message-passing system with lossy channels. The broadcast nature of the radio signal allows certain performance gains. In implementing $\mathcal{GDP}$ in this environment the concern is to preserve its correctness and termination properties. We discuss the modifications to preserve the algorithm's correctness first. Note that in order to satisfy non-trivial liveness properties we assume that our environment conforms to *transmission fairness*: if a process attempts to send infinitely many messages, all of its communication neighbors will receive infinitely many of them. This assumption is weaker than what is previously used for self-stabilizing algorithms in sensor networks [Herman and Tixeuil 2004; Mitton et al. 2006; Mitton et al. 2008]: it is usually assumed that the expected message transmission time for one hop neighbors is constant. Our idea is to use the timeouts such that the lost messages are recovered. To prove correctness of such an algorithm in asynchronous system model we would have to use an abstract form of timeouts (see [Gouda and Multari 1991] or [Gouda 1998]) that are enabled when the channel is empty.

There are two phases where the message recovery is important: request and release propagation. In case of request propagation, when the parent changes its state to **req**, it sends a message to its children and starts a timeout. When the timeout expires, the parent resubmits the request. Upon the receipt of the request, the child's actions differ depending on its state. As in the original algorithm, in case the child is in **idle**, it switches to **req** and further propagates the request; similarly, if the child is in **req**, it ignores the request. In case the child is in **rep**, it sends back the message informing the parent of its state. These actions ensure that the request will be propagated along the routing tree and the reply will be collected. As an efficiency optimization, a child may acknowledge the request message from its parent. This acknowledgment is done either explicitly or by broadcasting its own request to its children. The parent then resubmits its request only to the children that have not acknowledged it yet. Recall that for release propagation, the parent needs to ascertain that its children are **idle** before switching to **req** and starting to propagate the next request. Similar to the case of request propagation, the parent has to keep the list of its non-idle children and keep informing its children of its idle state until all of its children acknowledge (explicitly or implicitly) that they also switched to **idle**. When all its children are **idle** the parent can turn of its notification timeout.

Let us now address termination preservation of $\mathcal{GDP}$ in wireless networks. Note that co-satisfaction of stabilization and termination in message-passing systems is a rather difficult objective. In message passing, incorrect state may not be detected by any process unless they exchange messages. This means that all terminal states have to be legitimate. However, [Arora and Nesterenko 2005] demonstrate that mutual exclusion and, by extension, diners admits a solution with both of these properties. Notice that, as described, it is possible that the algorithm refined to operate in wireless sensor networks starts in an illegitimate fixpoint state where some child is in **rep** and its parent is in **idle**. That is, the state does not have actions enabled unless the environment requests the CS. This state is illegitimate: if there is a further request and the parent switches to **req**, then the parent may mistake the child's reply as the answer to its new request. This mistake may result in a safety violation (see [Arora and Nesterenko 2005] for a detailed discussion of this issue). A stabilizing algorithm cannot terminate in an illegitimate state. Thus, this particular terminal state has to be eliminated. The mechanism is as follows. If a process is in **req**, it periodically informs its parent about its state. If parent is in **idle**, it messages back with its state and forces the child to switch to **idle** as well. With this modification, the only terminal state is the one where every process is in **idle**. This is a legitimate state and our algorithm remains terminating and stabilizing.

In a wireless network of resource constrained devices, maintaining information of multiple conflict trees at each node may not be feasible. Note however, that much of it may be reused. In the extreme case, all nodes may share the same global spanning tree to reach conflict neighbors.

5. FURTHER EXTENSIONS

Extension to generic drinking philosophers. In the classic drinking philosophers problem, the set of conflict neighbors for each process p may vary with each CS access request. This set is the *conflict list.* This problem can be extended to the generic case of conflict neighbors. In this generalized drinking philosophers problem, the two processes are prevented from concurrently entering the CS only if one was the member of the conflict list of the other. $\mathcal{GDP}$, in its turn, can be extended to solve this generic drinking philosophers problem as well. In this extension each process p maintains the tree to the union of all its possible conflict list neighbors. In the simplest implementation, each request carries the complete conflict list along the spanning tree. The conflict neighbor q of p is prevented from entering the CS only if q is in the list. The algorithm can be further optimized in case the conflict lists are significantly shorter than the total set of possible conflict neighbors. Each process q in the tree maintains the set of all its descendants. Thus, p has the list of all its potential conflict neighbors. When p requests the CS, it propagates its complete conflict list. The child of p propagates the request only if it has a descendant in this set. Moreover, only the part of the list that is relevant to the particular branch is propagated. This process of selective propagation repeats in each node of the tree.

Extension to generic committee coordination problem. *The committee coordination problem* [Chandy and Misra 1988] is another resource allocation problem stated as follows. Rather than have a set of conflict neighbors, each process (professor) has a fixed set of *committee colleagues.* The process periodically requests to attend a committee and starts waiting. The process does care which committee to attend. A committee meeting may start only after all committee members are waiting. A committee meeting may only last a finite amount of time. A solution to the committee coordination problem has to satisfy two properties. The *safety property* requires that a process may not attend two committees at the same time; the *liveness* states that if all members of some committee are waiting then one of the members eventually attends one of the committees it is a member of.

This problem can be generalized similarly to the diners and drinkers. The solution is as follows. The conflicting entities: the committees are represented as philosophers. Two philosophers are conflict neighbors if their corresponding committees share a professor. Therefore, if one of the committees meets (i.e. the philosopher eats) then another committee with the same professor cannot meet. This satisfies the safety property of the committee coordination problem. To satisfy the liveness property, the professor's willingness to attend a committee has to be communicated to all committees in which this professor participates.

Observe that in the generalized version of the problem, the professors may be located multiple hops away from their committees. The communication mechanism is as follows. Each professor maintains a spanning tree (similar to the generic dining philosophers tree) to all its committees. Similarly, each committee maintains a separate spanning tree to all of its professors. The professor indicates its willingness to attend the committee by setting its state to **req**. This request propagates to all of its committees along the spanning tree. If all committee members are willing to attend the committee, the committee sets *needcs* to **true**. Then the synchronization with other committees proceeds as in the generalized version of the diners. After the CS entry, the committee notifies the professors through its own spanning tree. Thus the liveness property of the committee coordination property is satisfied.

Simplification to unfair case. Notice that some problems [Gradinariu and Tixeuil 2007], such as distance-k vertex coloring, maximal irredundant sets, etc. [Goddard et al. 2006] do not require fairness of CS access specified by the diners: in any computation of such a problem there are only finitely many CS accesses. If $\mathcal{GDP}$ is to be used for such a problem, it can be simplified. In the unfair case, an idle higher priority process does not have to wait for a lower priority neighbor. This obviates the need for *YIELD* and simplifies actions *stop*, *enter* and *join*. Moreover, the computations of such a program are finite. Thus, this program is capable of operating without the weak fairness assumption about action execution.

Future research directions. It is unclear if $\mathcal{GDP}$ is an optimal solution to generic diners with respect to space complexity. If the communication topology is dense, statically maintaining spanning trees may be expensive. Hence, the construction of a more space-efficient algorithm is an attractive area of future research. Observe that our algorithm does not tolerate crash faults. Naturally, any solution to diners requires that if two conflict neighbors requires the CS, one has to wait for the other. Our algorithm allows the formation of such waiting chains of arbitrary length. If a process crashes inside the CS, all processes that wait for it may not proceed. A stabilizing solution to classic diners that bounds waiting chains is known [Nesterenko and Arora 2002a]. It would be interesting to investigate if the length of the waiting chains in $\mathcal{GDP}$ can be limited using similar techniques.

Observe that the information propagation mechanism used in $\mathcal{GDP}$ resembles information propagation with feedback (PIF) *snap-stabilizing* algorithms [Blin et al. 2003; Bui et al. 1999]. Snap-stabilizing algorithms are a subclass of the self-stabilizing algorithms that have stricter safety properties. In the case of PIF, a snap-stabilizing stabilizing algorithm stabilizes in a single request propagation round. It would be interesting to investigate if our whole algorithm can be made snap-stabilizing.

In our algorithm, $\mathcal{GDP}$ assumes the existence of a spanning tree in the conflict neighborhood. In [Blin et al. 2003], the PIF constructs such a tree during the request propagation. It will be interesting to study if this extra flexibility of tree construction can be built into $\mathcal{GDP}$.

Rather than solving the generic diners directly, a solution may be compartmentalized as follows. For each process, the set of conflict neighbors can be treated as a network of peers overlayed on the communication neighbor network. Since the communication to the conflict neighbors is relatively uniform: each process sends request, receives reply and then sends release, this communication can be fashioned as multicast in overlay networks. An area of reliable multicast in overlay networks have been extensively studied [Banerjee et al. 2003; Shi and Turner 2002]. Using this as a primitive, it may be possible to use a solution to classic diners to solve generic diners.

An attractive property of a synchronization algorithm is *abortability* [Jayanti 2003; Scott and Scherer, III 2001]. An algorithm is abortable if a process can rescind its request to enter the CS within a bounded number of its own steps without actually obtaining the CS. It appears that $\mathcal{GDP}$ can be modified to incorporate this property. However, a definitive answer requires further research

REFERENCES

Arora, A. and Nesterenko, M. 2005. Unifying stabilization and termination in message-passing systems. *Distributed Computing 17,* 3 (March), 279–290.

Arumugam, M. and Kulkarni, S. 2005. Self-stabilizing deterministic TDMA for sensor networks. Tech. Rep. MSU-CSE-05-19, Michigan State University.

Banerjee, S., Kommareddy, C., Kar, K., Bhattacharjee, S., and Khuller, S. 2003. Construction of an efficient overlay multicast infrastructure for real-time applications. In *Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*. 1521–1531.

Beauquier, J., Datta, A., Gradinariu, M., and Magniette, F. 2000. Self-stabilizing local mutual exclusion and daemon refinement. In *12th International Symposium on Distributed Computing*. Springer, 223–237.

Blin, L., Cournier, A., and Villain, V. 2003. An improved snap-stabilizing PIF algorithm. In *6th International Symposium on Self-Stabilizing Systems, (SSS 2003)*, S.-T. Huang and T. Herman, Eds. Lecture Notes in Computer Science, vol. 2704. Springer, 199–214.

Boulinier, C., Petit, F., and Villain, V. 2004. When graph theory helps self-stabilization. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. ACM Press, 150–159.

Bui, A., Datta, A., Petit, F., and Villain, V. 1999. State-optimal snap-stabilizing PIF in tree networks. In *Forth Workshop on Self-Stabilizing Systems*. IEEE, 78–85.

Cantarell, S., Datta, A., and Petit, F. 2003. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *6th International Symposium on Self-Stabilizing Systems*. LNCS, vol. 2704. Springer, 102–112.

Chandy, K. and Misra, J. 1984. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems 6,* 4 (Oct.), 632–646.

Chandy, K. and Misra, J. 1988. *Parallel Program Design: a Foundation.* Addison-Wesley, Reading, Mass.

Datta, A., Gradinariu, M., and Raynal, M. 2005. Stabilizing mobile philosophers. *Information Procesing Letters 95,* 1, 299–306.

Dijkstra, E. 1968. *Cooperating Sequential Processes.* Academic Press.

Dolev, S. 2000. *Self-Stabilization.* MIT Press.

G, G. A. and Srimani, P. 1999. A self-stabilizing distributed algorithm to find the median of a tree graph. *Journal of Computer and System Sciences 58*, 215–221.

Gairing, M., Goddard, W., Hedetniemi, S., Kristiansen, P., and McRae, A. 2004. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters 14,* 3-4, 387–398.

Goddard, W., Hedetniemi, S., Jacobs, D., and Trevisan, V. 2006. Distance-K information in self-stabilizing algorithms. In *Proceedings of the 13th Colloquium on Structural Information and Communication Complexity (SIROCCO)*. 349–356.

Gouda, M. 1998. *Elements of Network Protocol Design.* John Wiley & Sons, Inc.

Gouda, M. and Haddix, F. 1999. The alternator. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*. IEEE Computer Society, 48–53.

Gouda, M. and Multari, N. 1991. Stabilizing communication protocols. *IEEE Transactions on Computers 40*, 448–458.

Gradinariu, M. and Tixeuil, S. 2007. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2007)*. IEEE, 46.

Herman, T. 1991. Adaptivity through distributed convergence. Ph.D. thesis, Department of Computer Science, University of Texas at Austin.

Herman, T. and Tixeuil, S. 2004. A distributed TDMA slot assignment algorithm for wireless sensor networks. In *Proceedings of the First International Workshop on Algorithmic Aspects of Wireless Sensor Networks*. 45–58.

Hoover, H. 1995. On the self-stabilization of processors with continuous states. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*. 8.1–8.15.

Huang, S. 2000. The fuzzy philosophers. In *Proceedings of the 15th IPDPS 2000 Workshops*, J. R. et al., Ed. Lecture Notes in Computer Science, vol. 1800. Cancun, Mexico, 130–136.

Jayanti, P. 2003. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC-03)*. 295–304.

Johnen, C., Alima, L., Datta, A., and Tixeuil, S. 2002. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters 12,* 3-4, 327–340.

Kulkarni, S. and Arumugam, M. 2003. Collision-free communication in sensor networks. In *Proceedings of the Symposium on Self-Stabilizing Systems (SSS), Springer-Verlag LNCS:2704*. 17–31.

Lynch, N. 1980. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, (STOC)*. ACM Press, 70–81.

Malhotra, M., Krasniewski, M., Yang, C., Bagchi, S., and Chappbell, W. 2005. Location estimation in ad-hoc networks with directional antennas. In *the 25th IEEE International Conference on Distributed Computing Systems*. 633–642.

Mitton, N., Fleury, E., Guérin-Lassous, I., Sericola, B., and Tixeuil, S. 2006. Fast convergence in self-stabilizing wireless networks. In *ICPADS*. IEEE Computer Society, 31–38.

Mitton, N., Paroux, K., Sericola, B., and Tixeuil, S. 2008. Ascending runs in dependent uniformly distributed random variables: Application to wireless networks. *Methodology and Computing in Applied Probability*.

Mizuno, M. and Nesterenko, M. 1998. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters 66,* 6, 285–290.

Nesterenko, M. and Arora, A. 2001. Self-stabilizing routing in wireless embedded. In *SRDS Workshop on Reliability in Embedded Systems*. New Orleans, LA, 16–22.

Nesterenko, M. and Arora, A. 2002a. Dining philosophers that tolerate malicious crashes. In *22nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 191–198.

Nesterenko, M. and Arora, A. 2002b. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing 62,* 5, 766–791.

Scott, M. L. and Scherer, III, W. N. 2001. Scalable queue-based spin locks with timeout. In *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01), ACM SIGPLAN Notices (8th PPOPP'2001)*. 44–52.

Shi, S. and Turner, J. S. 2002. Routing in overlay multicast networks. In *Joint Conference of the IEEE Computer and Communications Society (INFOCOM)*. Proceedings IEEE INFOCOM 2002, vol. 3. IEEE Computer Society, Piscataway, NJ, USA, 1200–1208.

Sivilotti, P., Pike, S., and Sridhar, N. 2000. A new distributed resource-allocation algorithm with optimal failure locality. In *Proceedings of the 12th IASTED International Conference on Parallel and Distributed Computing and Systems*. Vol. 2. IASTED/ACTA Press, 524–529.