Self-Stabilizing Philosophers with Generic Conflicts

Praveen Danturi, Mikhail Nesterenko*
Department of Computer Science
Kent State University
Kent, OH 44240 USA
{pdanturi, mikhail}@cs.kent.edu

Sébastien Tixeuil†
LRI-CNRS UMR 8623 &
INRIA Grand Large
Université Paris Sud, France
tixeuil@lri.fr

August 21, 2005

Dept. Of Computer Science,
Kent State University
Technical Report: TR-KSU-CS-2005-05

Abstract

We generalize the classic dining philosophers problem to allow critical section entry conflicts between non-neighbor processes. This generalization is motivated by a number of practical problems in distributed systems. We describe a deterministic self-stabilizing solution to the new problem. We extend our solution to handle a similarly generalized drinking philosophers problem.

1 Introduction

Self-stabilization (or just stabilization) [7, 10] is an elegant approach to forward recovery from transient faults as well as initializing a large-scale system. In this paper we present a stabilizing solution to our generalization of the dining philosophers problem.

The dining philosophers problem (*diners* for short) is a fundamental resource allocation problem [6]. The diners, as well as its generalization — the drinking philosophers problem [5], has a variety of applications. In diners, a set of processes (philosophers) periodically 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 his neighbors: it cannot share the CS with any of them. In spite of the conflict, 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 set of communication neighbors fully contains the set of conflict neighbors.

*This author was supported in part by DARPA contract OSU-RF #F33615-01-C-1901 and by NSF CAREER Award 0347485.

†This author was supported in part by the FNS grants FRAGILE and SR2I from ACI "Sécurité et Informatique". Some of the research for this paper was done while the author was visiting Kent State University.

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 TDMA slot assignment, cluster formation and routing backbone maintenance can be considered as instances of resource allocation problems. Yet, due to radio propagation peculiarities, the signal's interference range may exceed its effective communication range. Moreover, radio networks have so called hidden terminal problem. The problem is as follows. Let two transmitters t_1 and t_2 be mutually out of reception range, while 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. Such transmitters can be modeled as conflict neighbors that are not communication neighbors.

To accommodate the applications of this kind, we generalize the diners as follows. Instead of one, each process has two sets of neighbors: the conflict neighbors and the communication neighbors. These two sets are not, in general, related. The only restriction is that each conflict-neighbor has to be reachable through the communication neighbors.

Most solutions to classic diners can potentially be extended to this generalized problem. Indeed, if a separate communication channel is established to each conflict neighbor the classic diners program can be directly 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. However, 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, communicating the same message to each conflict neighbor separately is not efficient. This motivates our search for an effective solution to the generalized diners.

Related work. There exist a number of deterministic stabilizing solutions to classic diners [1, 3, 4, 9, 12, 13, 14, 15]. None of these solutions separate conflict and communication neighbors.

Meanwhile, researchers studied the problems that require such separation. Herman and Tixeuil [11] present a probabilistic TDMA slot assignment algorithm for wireless sensor networks. Arumugam and Kulkarni [2] propose a deterministic solution to the same problem. Gairing et al [8] propose an interesting stabilizing program for conflict neighbor sets containing the communication neighbors of distance at most two. They apply their program to a number of graph-theoretical problems. However, their program cannot solve the diners as it is not designed to allow each requesting process to enter the CS if its neighbors also continue requesting. That is, their program allows unfair computations.

Our contribution and paper outline. We generalize the diners problem to separate the conflict and communication neighbor sets of each process. We define this generalization and describe our notation and execution model in Section 2. In Section 3 we present a self-stabilizing deterministic solution to the problem for the case where the conflict set of each process contains its communication neighbors within a fixed distance in communication topology. We call this program $\mathcal{KDP}$. We provide a formal correctness proof of $\mathcal{KDP}$ in Section 4. We then estimate the stabilization performance of $\mathcal{KDP}$ and describe a number of extensions to $\mathcal{KDP}$ in Section 5. We generalize $\mathcal{KDP}$ to handle arbitrary conflict neighbor sets, solve generalized drinking philosophers, and simplify our solution to handle problems that do not require fairness of CS access. We show that stabilization time of $\mathcal{KDP}$ is independent of the system size.

2 Preliminaries

Program model. A program consists of a set of processes. Two binary relations N and M are defined over these processes. Processes p and q are *communication neighbors* if $(p,q) \in N$, and *conflict neighbors* if $(p,q) \in M$. 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 its 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 either the computation is infinite or it terminates in as state where none of the actions are enabled. 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**. 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 iff **true** converges to R.

Problem statement. The instance of the k-hop diners defines a set of processes, and for each process p — an arbitrary set of communication neighbor processes $N.p$. A set $M.p$ of *conflict neighbors* of p contains the the processes whose distance to p in the graph formed by communication topology is no more than k. Throughout the computation each process can request CS access an arbitrary number of times: from zero to infinity. A program that solves the k-hop 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, it is eventually allowed to do so.

3 $\mathcal{KDP}$ Algorithm 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 $M.p$. This tree is rooted in p. A stabilizing breadth-first construction of the tree is a relatively simple task [7]. Notice that the individual tree construction is independent of other trees constructions. Thus, these trees can be built in parallel. We assume that the tree construction is completed and the tree remains unchanged throughout the computation of the algorithm.

process p
const
M: k-hop neighborhood of p
N: immediate 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 k-hop 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: $state.(dad.p.r).r = \mathbf{req} \wedge state.p.r = \mathbf{idle} \wedge$
$((KIDS.p.r = \varnothing) \vee (\forall q : q \in KIDS.p.r : state.q.r = \mathbf{idle})) \longrightarrow$
$state.p.r := \mathbf{req}$

$[]$

back: $state.(dad.p.r).r = \mathbf{req} \wedge state.p.r = \mathbf{req} \wedge$
$((KIDS.p.r = \varnothing) \vee (\forall q : q \in KIDS.p.r : state.q.r = \mathbf{rep})) \vee$
$state.(dad.p.r).r = \mathbf{rep} \wedge state.p.r \neq \mathbf{rep} \longrightarrow$
$state.p.r := \mathbf{rep}$

$[]$

stop: $state.(dad.p.r).r = \mathbf{idle} \wedge$
$(state.p.r \neq \mathbf{idle} \vee r \in YIELD) \longrightarrow$
$YIELD := YIELD \setminus \{r\}$,
$state.p.r := \mathbf{idle}$

$]$

Figure 1: Process of $\mathcal{KDP}$

The processes in $M.p$ propagate CS request of its root along this tree. The request reflects from the leaves and, in the form of reply goes back to the root. When the root receives this reply, the root knows that its conflict neighbors are notified of its request.

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. Process p then waits until all these processes access the CS before requesting it again.

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 the state of 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 the 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 conflict neighbors.

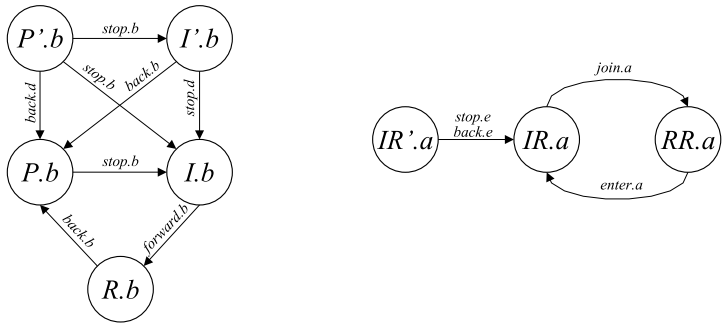
Action *join* states that p requests the CS when the application variable *needcs* is **true**, p itself, as well as its children in its own spanning tree, is idle and there are no lower priority conflict neighbors to wait for. As action *enter* describes, p enters the CS when its children reply and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

Action *forward* describes the propagation of 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 $\mathcal{KDP}$. 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 wait before initiating another request.

4 $\mathcal{KDP}$ Correctness Proof

Proof outline. We present the proof of correctness of $\mathcal{KDP}$ as follows. We first state a predicate we call $InvK$ and demonstrate that $\mathcal{KDP}$ stabilizes to it in Theorem 1. We then proceed to show that if $InvK$ holds, then $\mathcal{KDP}$ satisfies the safety and liveness properties of the 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. That is, we implicitly assume that a is universally quantified over all processes in the system. In particular, we focus on a child e of a, a descendant of a — b, b's parent c and one of b's children d. That is $e \in KIDS.a.a$, $b \in M.a$, $c \equiv dad.b.a$ and $d \in KIDS.b.a$.



i) intermediate process b if Inv holds for ancestors

ii) root process a

Figure 2: State transitions for an individual process

Since, we will be discussing 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 $state.b.a$. 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 two 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.

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

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

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

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

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

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 $M.a$.

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

To aid in exposition, we mapped the states and transitions for individual processes in Figure 2. 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 the primed version of R does not exist if $Inv.c$ holds for b's parent c. Indeed, to violate R, b should be requesting while c is either idle or replying. However, if $Inv.c$ holds, when c is in either of these two states, b cannot be requesting.

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 $Inv.a$ always holds if a is

requesting. The state transitions in Figure 2 are labeled by actions whose execution effects them. Loopback transitions are not shown.

Theorem 1 (Stabilization) *Program $\mathcal{KDP}$ stabilizes to $InvK$.*

Proof: By the definition of stabilization, $InvK$ should be closed with respect to the execution of the actions of $\mathcal{KDP}$, and $\mathcal{KDP}$ should converge to $InvK$. 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 $InvK$. 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 predicate holds for all its descendants. Naturally, the closure of $InvK$ follows.

By definition of a closure of a predicate, 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 a 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. If *enter.a* is enabled, each child of a is replying. Hence, when it is executed and sets the state of a to **idle**, $Inv.a$ holds. If *forward.e* is enabled, a is requesting. Thus, 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.

Let us 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 $Inv.c$ holds. If $I.b$ holds, *forward.c* it the only possible enabled action. If it is enabled, b is idle. The execution of *forward.c* sets 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* and *back.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 $P.b$.

Let us now examine the actions of d. If $I.b$ holds, the only possibly enabled action is *stop.d*. The execution of this action moves 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 $InvK$.

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 that affect the state of a are disabled. Moreover, for every child of a that is idle, all relevant actions are disabled as well. For the child e 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 $Inv.a$. Let also every descendant process f of a up to depth i confirm to $Inv.f.a$. Let the distance from a to b be $i+1$. We shall show that $Inv.b.a$ eventually holds. Notice that according to the proceeding closure proof, conjunction of $Inv.a$ and for each process f in the distance no more than i the predicate $Inv.f$ is closed.

Note that according to Figure 2, 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 state 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. When this action is executed for every requesting child of b, b leaves $I'.b$.

Suppose $b \in P'.b$. This means that there exists at least one child d of b that is not requesting. However, for every such process d, $back.d$ is enabled. When it is executed for every such process, b leaves $P'.b$.

Hence, $\mathcal{KDP}$ converges to $InvK$. □

Theorem 2 (Safety) *If $InvK$ 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 $InvK$, this means that every descendant of a is also replying. Thus, for every process x whose priority is lower than a's priority, $enter.x$ is disabled.

Note also, that since $enter.a$ is enabled, for every process y whose priority is higher than a's, $state.a.y$ is **idle**. According to $InvK$, 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 $InvK$ 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 will not be 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 $InvK$, 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 continues to be 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 $InvK$ 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 2). Thus, we have to prove 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 InvK 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 $i+1$. 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. This claim holds for the descendants up to depth i. Let b's distance to a be $i+1$.

When $InvK$ holds, the only way for b's parent c to leave **idle** is to execute *forward.c* (see Figure 2). Similarly, the only way for c to return to **idle** is to execute *stop.c* while c is replying [1].

However, *forward.c* is enabled only when b is idle. Also, according to $InvK$ 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 InvK holds, process a is requesting and a's priority is the highest among requesting processes in M.a 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 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 InvK 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 $InvK$, 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 other higher priority process $b \in M.a$ which is also requesting then, by Lemma 4, a eventually enters the CS.

[1] The argument is slightly different for a as it executes *join.a* and *enter.a* instead.

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 CS.

Suppose every higher priority process b enters the CS infinitely often. Since a is requesting, $state.b.a =$ **rep**. When b executes the CS, it enters a into $YIELD.b$. We assume that b enters the CS infinitely often. However, b can request the CS again only when $YIELD.b$ is empty. The only action that takes a out of $YIELD.b$ is $stop.b$. However, this action is enabled when $state.b.a$ is **idle**. Notice that, if $InvK$ 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 $InvK$ holds and a process a wishes to enter the CS, a eventually requests.*

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

Let us consider a process $b \in YIELD.a$. 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, $state.a.b$ is idle. If this is the case $state.a$ is enabled. When it is executed b is removed from $YIELD.a$.

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, $state.a.b$ is idle infinitely often. Moreover, a moves to idle by executing $stop.a$, which removes b from $YIELD.a$. The lemma follows. □

The theorem below follows from Lemmas 5 and 6.

Theorem 3 (Liveness) *If $InvK$ holds, a process wishing to enter the CS is eventually allowed to do so.*

This theorem concludes the correctness proof of $\mathcal{KDP}$.

5 Conclusion

Stabilization time and locality. Observe (see Figure 2) that each process executes at most two of its own actions before satisfying the stabilization predicate. Each of these action executions may only be interleaved by the action execution of the process neighbors. Let δ be the maximum degree of a process. Since stabilization proceeds from the root, there could be at most $2(\delta+1)k$ executions of actions in the conflict neighborhood before it stabilizes. If δ is not related to the number of processes in the system, the stabilization time of $\mathcal{KDP}$ depends only on k and thus independent of the system size.

Notice that the stabilization of one conflict neighborhood is independent of stabilization of another. Thus, the spacial extent of the state corruption is at most $2k$.

Implementation considerations. In our $\mathcal{KDP}$ algorithm, the CS execution is shown as a single step in action *enter*. However, the CS entry and exit can be separated into two actions without compromising the properties of $\mathcal{KDP}$.

To simplify the exposition, we presented $\mathcal{KDP}$ in a rather abstract execution model. In our algorithm, we assumed that a process can atomically read the variables of all its communication neighbors and update its own. However, this may not be practical in reality. Nesterenko and Arora [15] described a self-stabilizing mechanism for atomicity refinement to a model where a process may read variables of a single neighbor or update its own. A similar refinement mechanism can be applied to $\mathcal{KDP}$. Notice that Nesterenko and Arora propose a further refinement to message-passing model. This refinement is applicable to $\mathcal{KDP}$ as well.

Extension to generic conflict neighbors. Notice that we presented $\mathcal{KDP}$ for the case of rather strictly defined conflict neighborhood. However, $\mathcal{KDP}$ can be extended to handle an arbitrary conflict neighborhood relation.

In this case, each process p still has to have a spanning tree to all its conflict neighbors. Notice that, unlike $\mathcal{KDP}$, it is possible that some conflict neighbor q is only reachable through a process r that is not a conflict neighbor of p. In this case, r is included in p's spanning tree. Process r still propagates the requests and replies along p's tree. However, r ignores the state of p for its own CS access. For instance, r never enters p in $YIELD.r$.

Extension to generic drinking philosophers. In generic drinking philosophers, a set of conflict neighbors for each process p may vary with each CS access. $\mathcal{KDP}$ can be extended to solve this problem as well. In this case, p has to construct a spanning tree to the union of all of its possible conflict neighbors. Each process q in the tree, has the list of all its descendants. Thus, p has the list of all its conflict neighbors. When p requests the CS, it advertises the list of conflict neighbors for this request. The child of p propagates the request only if has a descendant in this set. The process repeats at each node.

Simplification to unfair case. Notice that some problems, such as distance-k vertex coloring, do not require fairness of CS access specified by the diners: such a problem has only finitely many CS accesses throughout any computations. If $\mathcal{KDP}$ is to be used for such a problem, it can be simplified. In an unfair case, an idle higher-priority process does not have to wait for a lower-priority neighbor to execute the CS before requesting itself. Hence, there is no need for YIELD. Which simplifies actions *stop*, *enter* and *join*. Moreover, the computations of such program are finite. Thus, this program is capable of operating without the weak fairness assumption on action execution.

Future research directions. It is unclear if $\mathcal{KDP}$ is an optimal solution to generalized diners with respect to space complexity. If a 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.

References

[1] G. Antonoiu and P.K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph that stabilizes using read/write atomicity. In *Proceedings of EuroPar'99*, volume 1685 of *Lecture Notes in Computer Science*, pages 823–830. Springer-Verlag, 1999.

[2] M. Arumugam and S.S. Kulkarni. Self-stabilizing deterministic tdma for sensor networks. Technical report, Michigan State University, 2005.

[3] J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium, Springer-Verlag LNCS:1914*, pages 223–237, 2000.

[4] C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 150–159, New York, NY, USA, 2004. ACM Press.

[5] K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[6] E. Dijkstra. *Cooperating Sequential Processes.* Academic Press, 1968.

[7] S. Dolev. *Self-Stabilization.* MIT Press, 2000.

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

[9] M.G. Gouda and F. Haddix. The alternator. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53. IEEE Computer Society, 1999.

[10] T. Herman. A comprehensive bibliography on self-stabilization (working paper). *CJTCS: Chicago Journal of Theoretical Computer Science*, 1995.

[11] T. Herman and S. Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. pages 45–58, 2004.

[12] S.T. Huang. The fuzzy philosophers. In J. Rolim et al., editor, *Proceedings of the 15th IPDPS 2000 Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 130–136, Cancun, Mexico, May 2000. Springer-Verlag.

[13] C. Johnen, L.O. Alima, A.K. Datta, and S. Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3-4):327–340, 2002.

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

[15] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.