SELF-STABILIZING PHILOSOPHERS WITH GENERIC CONFLICTS

A thesis submitted
to Kent State University in partial
fulfillment of the requirements for the

degree of Master of Science

by

Praveen Kumar Danturi

January, 2007

Praveen Kumar Danturi, M.S., January 2007 Computer Science

SELF-STABILIZING PHILOSOPHERS WITH GENERIC CONFLICTS

Director of Thesis: Dr.Mikhail Nesterenko

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. We present a self-stabilizing deterministic algorithm — $\mathcal{KDP}$ that solves a restricted version of the generalized problem where the conflict set for each process is limited to its k-hop neighborhood. We formally prove $\mathcal{KDP}$ correct and evaluate its performance. We then extend $\mathcal{KDP}$ to handle fully generalized problem. We further extend it to handle a similarly generalized drinking philosophers problem. We describe how $\mathcal{KDP}$ can be implemented in wireless sensor networks and demonstrate that this implementation does not jeopardize its correctness or termination properties.

Thesis written by
Praveen Kumar Danturi
B.S., Osmania University, 2003
M.S., Kent State University, 2007

Approved By

Dr. Mikhail Nesterenko_____, Advisor

Dr. Robert Walker_____, Chair, Department of Computer Science

Dr. John Stalvey_____, Dean, College of Arts and Sciences

Acknowledgement

I would like to take this opportunity to express my sincere gratitude to my advisor Dr. Mikhail Nesterenko for kindly providing guidance throughout my research work. I thank him for his continuous encouragement and making this thesis possible. It was a great pleasure to work on my thesis under his supervision. I also want to thank Dr. Sebastien Tixeuil for his valuable suggestions during his visit to Kent State University in 2005. His comments and suggestions have greatly helped in moulding this thesis to the way it is now.

Contents

List of Figures

Chapter 1

Introduction

Self-stabilization (or just stabilization) [17, 24] is an elegant approach to forward recovery from transient faults as well as initializing a large-scale system. The topic of Self-stabilization is described in more detail in Chapter 2. In this thesis we present a stabilizing solution to our generalization of the dining philosophers problem.

The dining philosophers problem (*diners* for short) [14] is a fundamental resource allocation problem. More information about the resource allocation problems is given in Chapter 3. The diners, as well as its generalization — the drinking philosophers problem [10], 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 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 effect. 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 (see for example [31]). 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. The only restriction is 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, communicating the same message to each conflict neighbor separately leads to excessive overhead. This motivates our search for an effective solution to the generalized diners.

Our contribution and thesis outline. In Chapter 2, we briefly discuss the topic of self-stabilization and significant research work in this area. In Chapter 3 we describe several resource allocation problems and the related work done in solving the diners. In Chapter 4, we discuss some of the papers which consider separation between conflict and communication neighbors. 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 Chapter 5. In Chapter 6, we present a self-stabilizing deterministic solution to a restricted version of this problem where the conflict set comprises the set of processes that are at most a fixed number of hops k away from the process. We call this program $\mathcal{KDP}$. To our knowledge, this algorithm is the first to solve the considered problem, even for the non-stabilizing case. In the same Chapter we provide a formal correctness proof of $\mathcal{KDP}$ and discuss its stabilization performance. We extend $\mathcal{KDP}$ to solve generalized diners in Chapter 7. In Chapter 8, we describe how $\mathcal{KDP}$ can be implemented in wireless sensor networks. We describe a number of further extensions to $\mathcal{KDP}$ in Chapter 9. Specifically, we generalize $\mathcal{KDP}$ to handle arbitrary conflict neighbor sets, as well as solve generalized drinking philosophers; we simplify our solution to handle problems that do not require fairness of the CS access. We end our discussion by explaining various applications of our $\mathcal{KDP}$ algorithm.

Chapter 2

Self-Stabilization

Dijkstra was the first to propose the concept of self-stabilization. Dijkstra [16] proposed that in distributed systems, independent of the initial global state, the local behaviour of the processes in the system can guarantee that the system satisfies a global predicate within a finite number of execution steps. In a distributed system processes do not have access to common storage containing the current global system state. Processes can exchange information only with their neighbors. However, the local actions executed by each process should result in achieving the common task. Hence, the property of self-stabilization is non-trivial to obtain.

Arora and Gouda [2] provide a foundation for self-stabilization program and define two properties *closure* and *convergence* to tolerate a class of faults. The *closure* property states that if a fault occurs when the system state is within the set of legal states, the resulting state is within some larger set and, if faults continue to occur, the system state remains within that larger set of states i.e, set of legitimate states is closed under system execution. The *convergence* property states that if faults stop occurring, the system eventually reaches a state within the legal set i.e starting from any system state, every system computation eventually reaches a legitimate state.

Dolev, Gouda and Schneider [18] propose a new class of self-stabilization called *silent* self-stabilization. Self-stabilizing algorithms are said to be *silent*, if after stabilization the values in the communication registers remain constant.

Herman [24] provides a comprehensive bibliography on self-stabilization. In his book on self-stabilization, Dolev [19] presents the fundamentals of self-stabilization and demonstrates the process of designing self-stabilizing distributed systems. He details the algorithms that can be started in an arbitrary state, allowing the system to recover from the faults that brought it to that state.

Chapter 3

Resource Allocation Problems

A resource allocation problem deals with scheduling the use of a shared resource between multiple competing processes in such a way that only one such process may access the shared resource at a time but eventually every process is allowed to access the resource. Below we describe the most well-known resource allocation problems.

3.1 The Mutual Exclusion Problem

The mutual exclusion problem deals with providing a mutually exclusive access to a single resource shared by multiple processes. The conditions that need to be satisfied are *safety* and *liveness*. The safety property states that only one process can access a shared resource at a time and the liveness property states that if a process requests access to a shared resource it should eventually be given a chance to do so. The process accessing the shared resource is said to be in the critical section (CS).

Several algorithms have been proposed earlier to solve the mutual exclusion problem. Lamport [30] and Ricart and Agarwala [36] proposed lock-based solutions to the distributed mutual exclusion problem where each process requesting to enter the CS needs to obtain permission from the other competing processes. Suzuki and Kasami

[37] propose a token-based solution to the same problem where a requesting process possessing the token can enter the CS. Nesterenko and Mizuno [35] present a self-stabilizing mutual exclusion algorithm based on Maekawa's algorithm.

3.2 Dining Philosophers Problem

The classic dining philosophers problem is a multiple process synchronization problem. The dining philosophers problem was first stated by Dijkstra [15]. He first proposed diners for a ring topology, in his example a group of philosophers sit around a circular table and a single fork is placed between each pair of adjacent philosophers as shown in the Figure 3.1. Each philosopher needs both forks: on his left and right hand side. The problem is to allow each philosopher to eat viz. to avoid deadlock and starvation. Deadlock is the condition where each philosopher cannot proceed as he is waiting for the other philosopher to release the fork. Starvation is the condition where a philosopher cannot have both the forks at the same time for an infinite time period. Other condition which needs to be considered is *fairness* condition. A solution is considered to be fair if each process gets an opportunity to access a shared resource as many times as its conflict neighbors. The diners problem was generalized to arbitrary topology where two or more processes may wish to execute the CS based on the conditions that only one requesting process among the competing processes access the CS at a time and each requesting process eventually executes the CS.

There exist a number of deterministic self-stabilizing solutions to diners [1, 6, 7, 23, 26, 27, 33, 34]. Antonoiu and Srimani [1] propose a solution to the mutual exclusion problem among neighboring nodes in a tree. They propose a serial model self

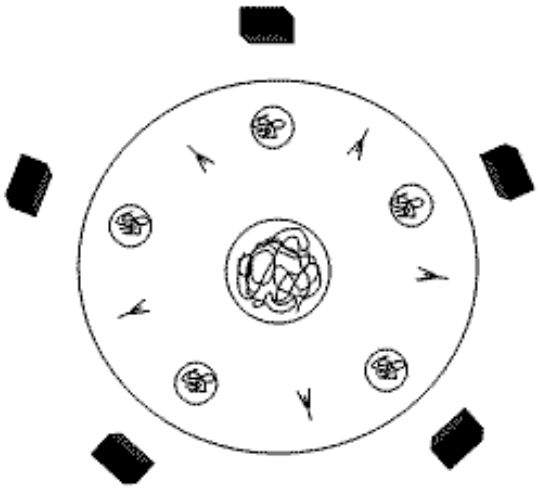

Figure 3.1: Diners table with five philosophers

stabilizing mutual exclusion algorithm that can be run in a distributed environment. No two nodes which are direct neighbors in the tree can execute the CS at the same time but two nodes which are farther than distance-1 can execute simultaneously. The proposed algorithm uses only bounded integers instead of the timestaps which are unbounded integers.

Gouda and Haddix [23] propose a solution to diners they call it an alternator. An alternator is a group of interacting processes that satisfy the conditions that neighboring process do not enter the CS at the same time in an arbitrary topology and also each action is executed infinitely often. Apart from this, the alternator is stabilizing to states where maximal concurrent actions are enabled. They also describe how alternators can be used in transforming a system stabilizing under serial execution to one that is stabilizing under concurrent execution. The transformations that en-

sure that conflicting actions are not executed simultaneously during any concurrent execution are given in their paper and [33]. Gouda and Haddix used bounded number of variables for the transformation and is also applicable to arbitrary topology where as in [33] Mizuno and Nesterenko used unbounded variables but it achieves the *silent* stabilization property. Kulkarni et al [28] propose a solution to alternators in read/write atomicity. Datta et al [13] solve a specific extension of diners.

3.3 Drinking Philosophers Problem

Chandy and Misra [10] genaralized diners to the drinking philosophers problem. Diners model deals with static resource-allocation scenarios, because each process has a fixed set of resources to acquire each time it becomes hungry whereas the drinking philosophers problem (*drinkers*) relaxes this constraint so that the required set of resources can change dynamically over the course of a computation. A drinking philosopher shares a number of distinct bottles with each neighbor and cycles through three states analogous to diners: tranquil, thirsty, and drinking. Each time a tranquil process becomes thirsty, it attempts to acquire some dynamically-determined subset of the bottles it shares with each neighbor. As such, diners is a special case of drinkers where the subset of bottles needed is always the entire set of bottles shared. An advantage of drinking philosophers is that unlike diners neighbors can drink simultaneously whenever they require disjoint subsets of bottles. This flexibility increases potential concurrency and overall resource utilization. The drinkers has known algorithms based on underlying dining solutions. Chandy and Misra [10] propose a fair solution to drinking philosophers problem using forks mechanism and distributed implementation of acyclic graphs. Cantarell et al [9] solve the drinking philosophers

problem. Nesterenko and Arora [34] generalize their solution to drinkers.

3.4 Committee Coordination Problem

Chandy and Misra [11] generalized diners to committee coordination problem. The problem is stated as follows: professors of some university organized themselves into committees where each commmittee can have one or more professors and each professor can be in zero or more committees. A professor starts by waiting to attend some committee meeting and remains waiting until a meeting of a committee of which he is a member is convened. A committee meeting is convened only if all the members of the committee are waiting and also no two committees can convene their meeting at the same time if they have a common member. Assuming that all meetings terminate in a finite time, the committeee coordination problem is to devise a protocol such that if all members of a committee are waiting then a meeting involving some member of this committee is convened. Bagrodia [5] presents a simple solution to this problem and develops a group of algorithms which use message counts to solve the synchronization problem and token circulation to solve the committee coordination problem. He also provides simulation results of the algorithms considering variations in topologies and conflicts in the system.

Chapter 4

Other Related Work

Classic diners assume that the conflict neighbors of a process are a subset of its communication neighbors. However, there are number of applications where the separation of conflict and communication sets will be useful.

TDMA slot assignment is the problem of assigning time slots to each process of a network in such a way that message collisions do not occur when processes transmit during their assigned time slots. Two processes a and b can transmit in the same time slot if they do not interfere with each other's communication: distance between a and b is greater than two, otherwise hidden terminal effect can occur. Hence, these transmitters can be modeled as conflict neighbors even though they may not directly communicate with each other.

A few studies [4, 25, 29] address the problem of self-stabilizing TDMA slot assignment. Herman and Tixeuil [25] 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 CSMA/CA mechanism that provides constant-time transmission with

high probability. The authors assume that the network is tightly synchronized so that the phases that use the CSMA/CA mechanism are clearly distinguished from the phases that use TDMA mechanism.

Arumugam and Kulkarni [4, 29] propose deterministic solutions to the same problem. In [4], they proposed a self-stabilizing deterministic solution to TDMA problem through a token passing mechanism. To avoid conflicts they propose to serialize channel assignments by circulating a single assignment token (privilege) throughout the network. A spanning tree is maintained in the network and a token is circulated by the root to all the processes in the network using existing graph traversal algorithms. The proposed TDMA algorithm is based on a distance-two coloring algorithm. The initial time slots are the colors assigned to the processes and their succesive time slots depend on the maximum degree of the network. Arumugam and Kulkarni first proposed a TDMA algorithm for a read/write model and then converted it to write all with collision model (WAC). Some optimizations for token circulation and recovery are proposed along with the improvement of bandwidth utilization. Addition or removal of a process does not violate the normal operation until the maximal degree of the tree is not increased by this addition. As the time slots are assigned serially in the network this process may not be efficient. Arumugam and Kulkarni [29] consider a regular grid topology where each node is aware of its position in the grid.

Gairing et al [20] propose an interesting stabilizing program for conflict neighbor sets containing the communication neighbors of distance at most two. They propose a mechanism that allows a node to act only on correct distance-two knowledge. They first propose an algorithm for 2-packing problem assuming that each node can read

the states of all nodes within distance two from it. A 2-packing algorithm gives a set of nodes where no two nodes in the set are adjacent and have no common neighbor in the given graph. Later Gairing et al [20] gave an algorithm to convert it to the model where each node can instataneously read the states of its direct neighbors only. The authors also showed how this algorithm can be applied to solve some graph-theoretical problems such as distance-two coloring and $\{k\}$-domination. Their algorithm uses process identifiers to give precedence and does not guarantee fairness and so it is not applicable to diners. Goddard et al [21] 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.

Huang [26] proposes two protocols to solve the self-stabilizing diners problem. The first one named as A-protocol has the self-stabilizing property only if the network is acyclic. So, it assumes that a directed network exists in a given network which is acyclic. In this protocol, each process maintains a single control bit which can hold the value of **0** or **1**. The later direction of the edge between two processes depends on the **XOR** of control bits of those processes. The process with all the edges directed towards it (sink) executes the CS. After executing the CS, process changes its control bit value which results in reversing the directions of edges of this process. This protocol assumes that the rules are non-interfering in the sense that once the guard is true it remains true until the action part is executed. The second protocol called B-protocol provides a self-stabilizing solution to diners for arbitrary networks. This protocol assumes that all links in the network are colored from a color set in such a way that all the sub-networks induced by links of each color (which can

be disjoint) are acyclic. Each process holds as many control bits as the number of different colors of its edges. This protocol does not assume that directed links of the complete network are acyclic. Each process executes the CS when it gets the sink status of all the sub-networks of which it is part of. This B-protocol also supports correct distributed execution of the serial correct self-stabiling protocols.

Mizuno and Nesterenko [33] present an algorithm for transforming self-stabilizing serial model programs to a self-stabilizing program that runs in an asynchronous shared memory parallel computing environment. This algorithm uses timestamps to guarantee mutually exclusive execution of guarded commands among neighbors and hence can be used as a weakly fair solution to the diners. This algorithm is applicable to arbitrary topology and also preserves the silent stabilization property.

Nesterenko and Arora [34] present a stabilization preserving atomicity refinement from a model where a neighbor can atomically access all the states of its neighbors and update its own state, to a model where a process can atomically access only the state of one of its neighbors or update its own state. A low atomicity, bounded space solution is proposed to the self-stabiling diners. This solution can easily be extended to solve drinkers problem and also can be refined to message passing model. Some other characteristics of their proposed refinement are *fixpoint-preserving*, meaning that terminating computations of high atomicity program correspond only to terminating actions of low atomicity program and also *fairness-preserving*, i.e, weakly-fair action execution of high-atomicity program is preserved in the refined low-atomicity model.

Chapter 5

Preliminaries

Program model. For the formal description of our program we use simplified UNITY notation [11, 22]. 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.

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. An instance of the generalized 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 $q \in N.p$. Same applies to $M.p$. Throughout the computation each process requests the CS access an arbitrary number of times: from zero to infinity. A program that solves the generalized 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.

A restriction of the generalized diners problem which we call *k-hop diners* [12] specifies that $M.p$ for each process p contains the processes whose distance to p in the graph formed by the communication topology is no more than k.

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

$[]$

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

$[]$

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

$]$

Figure 5.1: Process of $\mathcal{KDP}$

Chapter 6

$\mathcal{KDP}$ Algorithm

6.1 Description

Algorithm overview. The main idea of the algorithm is to coordinate the 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 a spanning tree is a relatively simple task [17].

The processes in this 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 [8].

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 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 the 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 the CS entry of p itself. The remaining three: *forward*, *back* and *stop* — propagate the 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* 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

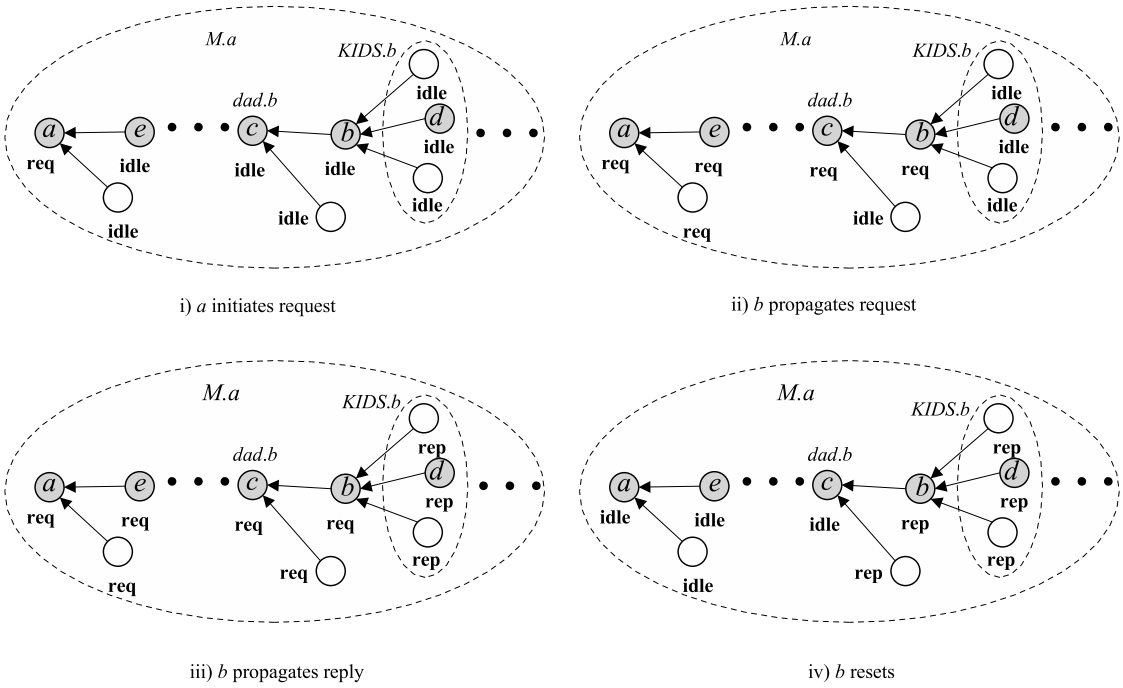

Figure 6.1: Phases of $\mathcal{KDP}$ operation

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 $\mathcal{KDP}$. 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 for before initiating another request.

[1]In Chapter 9, we demonstrate how to extend our algorithm to perform the CS entry and exit in separate actions.

Example operation. The operation of $\mathcal{KDP}$ in legitimate states is illustrated in Figure 6.1. 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 6.1, i). This request propagates to process b, which executes *forward* and sets $state.b.a$ to **req** as well (Figure 6.1, 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 6.1, iii). After the reply 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 6.1, iv).

6.2 Proof of Correctness

Proof outline. We present $\mathcal{KDP}$ correctness proof 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 k-hop 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 6.1). 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:

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

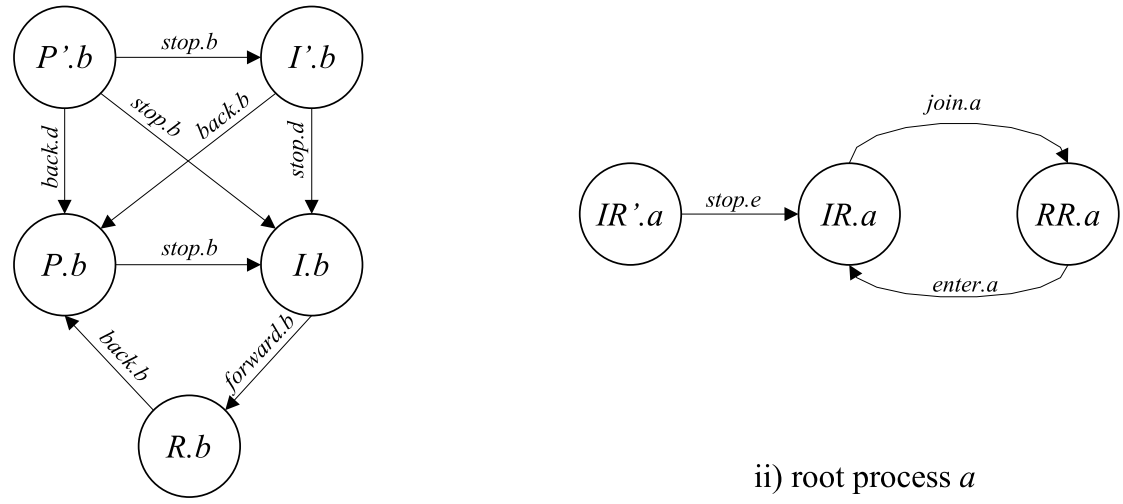

Figure 6.2: State transitions for an individual process

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 6.2. 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 and 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 since $state = \mathbf{req}$ falsifies the antecedent of $Inv.a$, the predicate always holds if a is requesting. The state transitions in Figure 6.2 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 predicates hold for all its ancestors. 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 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 $P.b$.

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 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 $InvK$.

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 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 $Inv.a$. Let also 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 6.2, 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. 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 $b \in P'.b$. This means that 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, $\mathcal{KDP}$ 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 6.2). 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 $i+1$ 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 $InvK$ holds, the only way for b's parent c to leave **idle** is to execute *forward.c* (see Figure 6.2). 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 $InvK$ 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 and process a is requesting such that and a's priority is the highest among the processes that ever request the CS 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 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 $InvK$ 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 higher priority process $b \in M.a$ which is also requesting then, by Lemma 4, a eventually enters the CS.

[2] The argument is slightly different for $c = a$ 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 there exists a higher priority process b that 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 if $YIELD.b$ is empty. The only action that takes a out of $YIELD.b$ is $stop.b$. However, this action is enabled if $state.b.a$ 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 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 a process to $YIELD$ only when it executes the CS. Thus, as a remains idle, processes can only be removed from $YIELD.a$.

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 $stop.a$ 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.

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

Corollary 1 Program $\mathcal{KDP}$ is a self-stabilizing solution to the k-hop diners problem.

6.3 Stabilization Efficiency Evaluation

Observe (see Figure 6.2) 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 $2k$. Notice also that the locality extends to the trees used by $\mathcal{KDP}$. The individual tree

construction is independent of construction of other trees. Thus, these trees can be built or stabilized in parallel.

Chapter 7

Solution to
Generalized Dining Philosophers

Notice that we presented $\mathcal{KDP}$ for the case of a rather strictly defined conflict neighborhood. However, $\mathcal{KDP}$ can be extended to handle an arbitrary symmetric 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 $YIELD.r$.

Notice, that it may happen that some branches of the constructed tree for some process of p do not contain its conflict neighbors at all. The CS request propagation from p to such a branch is not necessary. To avoid such propagation our program can be further optimized as follows. If a leaf of a tree is not a conflict neighbor of p, it so informs its parent. If process q does not have conflict neighbors of p in a certain

branch, q does not forward p's requests to that branch. If process q does not have any conflict neighbors of p at all among its descendants and q itself is not a conflict neighbor of p, q informs its parent about it. Thus, the tree is pruned to contain only p's conflict neighbors and their ancestors which further improves the efficiency of our program.

Chapter 8

Implementation in Wireless Sensor Networks

As we motivated $\mathcal{KDP}$ by the problems arising in wireless sensor networks, we would like to discuss implementing our algorithm in this environment. 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 $\mathcal{KDP}$ 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. Note that this assumption is weaker than used previously for self-stabilizing algorithms in sensor networks [25, 32]: 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. 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 the 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 $\mathcal{KDP}$. Note that co-satisfaction of stabilization and termination in message-passing systems is a rather difficult objective. However, Arora and Nesterenko [3] demonstrated 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 terminal state where some child is in **rep** and its parent is in **idle**. 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 [3] 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.

Chapter 9

Further Extensions

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 [34] 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 drinking philosophers. In the classic drinking philosophers problem, the set of conflict neighbors for each process p may vary with each CS access. This problem can be extended to the generic case of conflict neighbors in a straightforward manner.

$\mathcal{KDP}$ can be extended to solve the generalized drinking philosophers 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 potential conflict neighbors. When p requests the CS, it advertises the list of the actual conflict neighbors for this request. The child of p propagates the request only if it 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: in any computation of such a problem there are only finitely many CS accesses. If $\mathcal{KDP}$ 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 program are finite. Thus, this program is capable of operating without the weak fairness assumption about action execution.

Chapter 10

Applications

There are several applications of our algorithm besides the solution to generalized dining philosophers.

Distance k-coloring: distance k-coloring problem is to assign colors to each node in a graph such that no two nodes within distance k of each other have same color assigned to them using minimum number of colors possible. Our algorithm can be modified to make it applicable to distance k-coloring. Each process p maintains a set of colors that all distance k neighbors selected. This set can be read by its communication neighbors. Once the *enter* action is enabled p selects a minimum color available excluding the colors in its used color set. Another disjunct such as, if color of p matches with any of its used colors needs to be added to *join.p*. The actions *forward.p* and *stop.p* also needs to be modified. Along with the changes in state they also need to update the color variables if they differ from its parents color variables.

TDMA slot assignment: As discussed earlier, TDMA slot assignment requires distance-2 information of a network to avoid hidden terminal effect. Our $\mathcal{KDP}$ algo-

rithm can be applied to solve this problem by setting k to 2. From the safety property, only one process in the distance-2 neighborhood can enter the CS to avoid collisions. Arumugam and Kulkarni [4] propose a self-stabiling solution for TDMA in sensor networks where time slot assigned to each process is based on color associated to that process. The colors are assigned to every process by passing a single token through out the network in such a way that no two processes within distance-2 neighborhood get the same color. In our algorithm, a process which wants to transmit can request a time slot by changing its state to **req**. Moreover our algorithm provides partial fairness.

Minimal $\{k\}$ domination: A $\{k\}$ dominating set **S** is a subset of **V** of a graph G(**V**, **E**) where every node in **V-S** is atmost k-hops away from atleast one node in **S**. This dominating set is minimal when removal of any single node from **S** makes it a non dominating set. Our algorithm can be modified to find **S**. Each process maintains a variable indicating if a process in its k-hop neighborhood entered into **S**. A disjunct needs to be added to the guard of action *enter.p* as follows, check for the condition if no k-hop distance neighbor of p is in **S**. If the action *enter.p* is enabled then enter p into **S**. The actions *forward.p* and *stop.p* should be modified in such a way that neighbors update and propagate the information of p's entry into **S** upto k-hop neighbors of process p. Moreover the resulting dominating set is a Minimal.

2-packing: A 2-packing in a graph is a set **S** of nodes such that no two nodes in **S** are adjacent and no two nodes in **S** have a common neighbor. A 2-packing can be attained from our $\mathcal{KDP}$ algorithm by modifying it as discussed in the above Minimal

$\{k\}$ domination and setting value of k to 2.

Future research directions. It is unclear if $\mathcal{KDP}$ is an optimal solution to generalized 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. An other interesting work is to extend our solution to committee coordination problem.

Bibliography

[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] A. Arora and M.G. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[3] Anish Arora and Mikhail Nesterenko. Unifying stabilization and termination in message-passing systems. *Distrib. Comput.*, 17(3):279–290, 2005.

[4] M. Arumugam and S.S. Kulkarni. Self-stabilizing deterministic TDMA for sensor networks. Technical Report MSU-CSE-05-19, Michigan State University, 2005.

[5] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Trans. Softw. Eng.*, 15(9):1053–1065, 1989.

[6] 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.

[7] 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.

[8] A. Bui, A.K. Datta, F. Petit, and V. Villain. Space optimal PIF algorithm: self-stabilized with no extra space. In *IEEE International Conference on Performance, Computing and Communications*, pages 20–26, 1999.

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

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

[11] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Mass., 1988.

[12] Praveen Danturi, Mikhail Nesterenko, and Sébastien Tixeuil. Self-stabilizing philosophers with generic conflicts. In Ajoy K. Datta and Maria Gradinariu, editors, *Eighth International Symposium on Stabilization, Safety, and Security on Distributed Systems (SSS 2006)*, Lecture Notes in Computer Science, page to appear, Dallas, Texas, November 2006. Springer Verlag.

[13] A.K. Datta, M. Gradinariu, and M. Raynal. Stabilizing mobile philosophers. *Information Procesing Letters*, 95(1):299–306, 2005.

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

[15] E.W. Dijkstra. *Hierarchical Ordering of Sequential Processes*, pages 72–93. Academic Press, 1972.

[16] E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

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

[18] S. Dolev, M.G. Gouda, and M. Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[19] Shlomi Dolev. *Self-stabilization.* MIT Press, Cambridge, MA, USA, 2000.

[20] 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.

[21] W. Goddard, S.T. Hedetniemi, D.P Jacobs, and V Trevisan. Distance-k information in self-stabilizing algorithms. *To appear in the Proceedings of the 13th colloquium on Structural Information and Communication Complexity (SIROCCO06).*

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

[23] 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.

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

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

[26] 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.

[27] 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.

[28] S. S. Kulkarni, C. Bolen J. Oleszkiewicz, and A. Robinson. Alternators in read/write atomicity. *Information Processing Letters*, 2005. to appear.

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

[30] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–564, 1978.

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

[32] N. Mitton, E. Fleury, I. Guérin-Lassous, B. Séricola, and S. Tixeuil. On fast randomized colorings in sensor networks. Research Report LRI-1416, LRI, June 2005.

[33] 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.

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

[35] M. Nesterenko and M. Mizuno. A quorum-based self-stabilizing distributed mutual-exclusion algorithm. to appear in Journal of Parallel and Distoributed Computing http://www.cs.kent.edu/∼mikhail/Research/maekawa.ps.

[36] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[37] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3(4):344–349, 1985.