SCALABLE DESIGN OF FAULT-TOLERANCE FOR WIRELESS SENSOR NETWORKS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Murat Demirbas, M.S., B.S.

* * * * *

The Ohio State University

2004

Dissertation Committee:

Anish Arora, Adviser

Neelam Soundarajan

Paolo Sivilotti

Approved by

Adviser

Computer & Information Science

ABSTRACT

Since wireless sensor networks are inherently fault-prone and since their on-site maintenance is infeasible, scalable self-healing is crucial for enabling the deployment of large-scale sensor network applications. To achieve scalability of self-healing, in this dissertation we focus on addressing *(1)* the scalability of the cost-overhead of self-healing with respect to the size of the network, and *(2)* the scalability of the design effort for self-healing with respect to the size of the application software.

Our research on *fault-containment* addresses the first problem: By confining the contamination of faults within a small area, this approach achieves healing within work and time proportional to the size of the perturbation, independent of the size of the network. Our research on *specification-based design of self-healing* addresses the second problem: Since specifications are more succinct than implementations, this approach yields efficient design of self-healing even for large implementations.

These two research directions are complementary, and together enable a *scalable design of local self-healing for large-scale sensor network applications.*

To my family.

ACKNOWLEDGMENTS

I am indebted to several people for their help and support throughout my Ph.D. journey.

The most important among these is my adviser Anish Arora. I am especially thankful to him for his patience toward me. I will consider it a great accomplishment if a little bit of his dedication and perseverance rubbed off on me.

I am grateful to my lab-mates, Hongwei Zhang, Sandip Bapat, Vinod Krishnamurty, Vinayak Naik, and Bill Leal for their company and help.

I am also indebted to Paolo Sivilotti, Neelam Soundarajan, Bruce Weide, and several other faculty members for their mentoring.

Finally, I would like to thank Ebru and my parents for their love and support.

VITA

April 21, 1976	Born - Trabzon, Turkey
1997	B.S. Computer Science and Engineering, Middle East Technical University, Ankara, Turkey.
2000	M.S. Computer & Information Science, The Ohio State University.
September 2003	Advanced School on Mobile Computing, Pisa, Italy.
June-September 2001	IBM T.J. Watson Research Center, Hawthorne, NY.
1998-present	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

Research Publications

M. Demirbas and H. Ferhatosmanoglu. "Peer-to-Peer Spatial Queries in Sensor Networks." *The Third IEEE International Conference on Peer-to-Peer Computing*, Sweden, September 2003.

M. Demirbas, A. Arora, and M. Gouda. "A Pursuer-Evader Game for Sensor Networks." *Sixth Symposium on Self-Stabilizing Systems*, pages 1–16, San Francisco, June 2003.

M. Demirbas and A. Arora. "Convergence Refinement." *International Conference on Distributed Computing Systems (ICDCS'2002)*, Vienna, Austria, July 2002.

A. Arora, M. Demirbas, and S. S. Kulkarni. "Graybox Stabilization." *International Conference on Dependable Systems and Networks (DSN'2001)*, pages 389–398, Goteborg, Sweden, July 2001.

A. Arora, S. S. Kulkarni, and M. Demirbas. "Resettable Vector Clocks." *Nineteenth ACM Symposium on Principles of Distributed Computing (PODC)*, pages 269–278, Portland, July 2000.

P. A. G. Sivilotti and M. Demirbas. "Introducing Middle School Girls to Fault Tolerant Computing." *Technical Symposium on Computer Science Education (SIGCSE)*, February 2003.

M. Demirbas, A. Arora, and V. Mittal. "FLOC: A Fast Local Clustering Service for Wireless Sensor Networks." *Workshop on Dependability Issues in Wireless Ad Hoc Networks and Sensor Networks (DIWANS/DSN 2004)*, Florence, Italy, June 2004.

M. Demirbas, A. Arora, T. Nolte, and N. Lynch. "STALK: A Self-Stabilizing Hierarchical Tracking Service for Sensor Networks." *Technical Report, OSU-CISRC-4/03-TR19*, Ohio State University, April 2003.

M. Demirbas. "Resettable Vector Clocks: A Case Study in Designing Graybox Fault-tolerance." *MS Thesis, OSU-CISRC-4/00-TR11*, Ohio State University, February 2000.

FIELDS OF STUDY

Major Field: Computer & Information Science

Studies in:

Distributed systems and networks	Prof. Anish Arora
Fault-tolerant computing	Prof. Anish Arora
Wireless sensor networks	Prof. Anish Arora
Formal methods	Prof. Anish Arora
Component-based design	Prof. Anish Arora

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Fault-tolerance is the ability of a system to deliver a desired level of functionality in the presence of faults. Fault-tolerance is crucial for many systems and is becoming vitally important for computing- and communication- based systems as they become intimately connected to the world around them, using sensors and actuators to monitor and shape their physical surroundings.

In contrast to only 2% of processors that find their way into interactive computers, such as laptops, desktops, and servers, the remaining 98% of processors are employed in embedded computers, such as those used in cell phones, personal digital assistants, vehicles, robots, home and industrial appliances [103]. Various efforts are beginning to provide ubiquitous network connectivity for these embedded devices to harvest the information derived by these embedded nodes and to enable remote controlling of these embedded systems [91]. Owing to the rapid growth rate of the embedded systems market, these networked, embedded devices are expected to outnumber humans by a hundred or thousands to one in the near future [103].

A prime example of the rising popularity of embedded systems is the sensor networks [2, 54, 104]. Recent advances in embedded systems technology have made it feasible to build low-cost and low-power wireless sensor nodes, and have, hence,

enabled deployment of large-scale wireless sensor networks (with potentially many thousands of nodes). Even in this early stages of their development, sensor networks have already found several applications. They are employed in habitat monitoring to study the nesting behaviors of birds on a remote island [78], in precision agriculture to monitor the humidity levels at different parts of a vineyard [57], and in civil engineering to monitor the stress level of structures under earthquake simulations [62]. A major application area for sensor networks is the military and surveillance systems [7, 33, 35, 98].

Sensor networks introduce new challenges for fault-tolerance. Sensor networks are inherently fault-prone due to the shared wireless communication medium: message losses and corruptions (due to fading, collision, and hidden-node effect) are the norm rather than the exception. Moreover, node failures (due to crash and energy exhaustion) are commonplace. Thus, sensor nodes can lose synchrony and their programs can reach arbitrary states [59]. Since on-site maintenance is not feasible, sensor network applications should be self-healing. Another challenge for fault-tolerance is the energy-constraint of the sensor nodes. Applications that impose an excessive communication burden on nodes are not acceptable since they drain the battery power quickly. Thus, self-healing of sensor network applications should be local and communication-efficient.

1.1 Scalability of Fault-Tolerance for Sensor Network Applications

Since wireless sensor networks are inherently fault-prone and since their on-site maintenance is infeasible, scalable self-healing is crucial for enabling the deployment of large-scale sensor network applications. To achieve scalability of self-healing, in

this dissertation we focus on addressing *(1)* the scalability of the cost-overhead of self-healing with respect to the size of the network, and *(2)* the scalability of the design effort for self-healing with respect to the size of the application software.

1.1.1 Scalability with respect to network size

Several sensor network services, such as tracking, routing, and spatial querying, require continuous maintenance of distributed data structures, such as trees, paths, and clusters, over a large number of sensor nodes. This is a challenging task because message losses and corruptions (due to fading, collisions, and hidden node effects) and node failures (due to software/hardware crashes or energy exhaustion) can drive portions of these large-scale structures to be arbitrarily corrupted and hence to become inconsistent with the rest of the structure.

For dealing with arbitrary corruptions, we need self-healing systems: A self-healing system ensures eventual satisfaction of system specifications upon starting from a corrupted state. However, since faults can temporarily violate the program specifications in a self-healing system, extra care should be taken for containing the effects of faults: Faults in one part of the system may contaminate the entire system and hence may result in a high-cost, system-wide correction.

Thus, mechanisms for local containment of faults are needed for continuous and local maintenance of large-scale data structures.

1.1.2 Scalability with respect to implementation size

Since the complexity of software grows drastically with respect to its size, large-scale software systems are extremely error-prone and fail frequently. Especially for

sensor network applications, that are inherently distributed, reasoning about the system and verification of correctness are more difficult due to the lack of a centralized controller and the lack of a globally shared memory. Again due to their overwhelming complexity, design of fault-tolerance for large-scale software systems remains a challenging task.

Whitebox approaches for designing fault-tolerance, such as exception handling, forward recovery, recovery blocks [92], and application-specific fault-tolerance methods [10, 11], assume that the implementation is fully available, and study the source-code for designing fault-tolerance. However, they are not applicable for large-scale software systems because the task of studying the implementation and designing a fault-tolerant version becomes unbearable as the size of the implementation grows.

Blackbox solutions, such as reset and restartability approaches, may be adequate for centralized software systems, however they are inapplicable for massively distributed software since a reset of the software would mean a global reset of the entire network, and would incur a lot of work and down time.

Thus, a more efficient and informed approach is needed for achieving scalable design of fault-tolerance with respect to software size.

1.2 Thesis

In this dissertation, we address the above two scalability questions.

Towards addressing the scalability problem of cost-overhead of fault-tolerance with respect to the network size, we propose that **scalable design of fault-tolerance**

to distributed data structures can be achieved by using efficient and lightweight *fault-containment techniques for self-healing*. By confining the contamination of faults within a small area, this approach achieves *fault-local self-healing*: Work and time spent for recovery are proportional to the size of the perturbation as opposed to the size of the network.

Towards addressing the scalability problem of fault-tolerance design with respect to the implementation size, we propose that **scalable design of fault-tolerance can be achieved without knowledge of system implementation but with knowledge only of system specification**. That is, for the design of fault-tolerance we eschew knowledge of system implementation in favor of knowledge of system specification. Since specifications are typically more succinct than implementations, our *specification-based design of fault-tolerance* approach offers the promise of scalability when the design effort for adding fault-tolerance is proportional to the size of the system. Also, since specifications admit multiple implementations and since system components are often reused, a specification-based approach offers reusability. Finally, in contrast to a blackbox approach, a specification-based approach allows the design of efficient (low-cost) fault-tolerance by virtue of exploiting more information about the system.

We give a brief overview of these two techniques in the following two subsections.

1.2.1 Fault-containment for scalability with respect to network size

For achieving local self-healing of hierarchical tracking of mobile objects in sensor networks, we developed a hierarchy-based fault-containment technique [35]. The key idea of this technique is to wait for a longer time before updating a wider region's

view. We achieve this by using larger timeouts before propagating an update to the higher levels of the hierarchy. This way, more recent (more accurate) updates coming from lower levels can catch up to (contain) misinformed updates at higher levels. As a result, contamination due to faults is restricted to an area proportional to the perturbation size (i.e., the size of the initially faulty area), and our tracking path stabilizes in work and time proportional to the perturbation size instead of the network size. Furthermore, our solution is such that the latency imposed by waiting for larger timeouts at higher levels of the hierarchical partitioning does not affect the availability or quality of tracking; it is still possible to seamlessly track continuously moving objects.

For achieving local self-healing of clustering, we developed a stretch-factor based fault-containment technique [34]. The key idea of this technique is to allow each cluster to tolerate expansion up to two-fold of its ideal size. This way, the faults hitting a cluster are subsumed locally within that cluster, and cascading effects, that may require a re-clustering of the entire network, are avoided. For example, we show that thanks to the stretch-factor of two-fold, the nodes in a collapsed cluster can either join their neighboring clusters or form a new cluster without disturbing their neighboring clusters.

1.2.2 Specification-based design for scalability with respect to implementation size

In order to demonstrate that scalable design of fault-tolerance is achievable via a specification-based approach, we have developed a novel method that enables such a design. Loosely speaking, our method is to first design a *wrapper* component to add fault-tolerance at the system specification level and then to transform this wrapper to

the system implementation level by means of a *fault-tolerance preserving and compositional refinement* [6,32]. Even though the wrapper is designed solely by studying the system specification and not the system implementation, the nature of our transformation enables us to conclude that the transformed wrapper provides fault-tolerance to the system implementation.

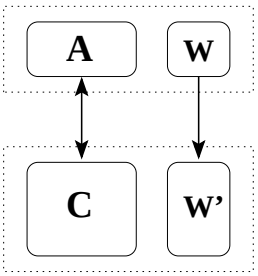


Figure 1.1: Specification-based design technique.

For example, given an abstract system specification $\mathcal{A}$, we first design a fault-tolerance wrapper $\mathcal{W}$ such that adding $\mathcal{W}$ to $\mathcal{A}$ yields a fault-tolerant system. Our transformations ensure that for any concrete implementation $\mathcal{C}$ of $\mathcal{A}$, adding a concrete implementation $\mathcal{W}'$ of $\mathcal{W}$ would also yield a fault-tolerant system.

Note that since the refinements from $\mathcal{A}$ to $\mathcal{C}$ and $\mathcal{W}$ to $\mathcal{W}'$ can be done independently, specification-based design enables *a posteriori* or dynamic addition of fault-tolerance. That is, given a concrete implementation $\mathcal{C}$, it is possible to add fault-tolerance to $\mathcal{C}$ by first designing an abstract fault-tolerance wrapper $\mathcal{W}$ using solely an abstract specification $\mathcal{A}$ of $\mathcal{C}$, and then adding a concrete refinement $\mathcal{W}'$ of $\mathcal{W}$ to $\mathcal{C}$.

1.3 Outline of the Dissertation

This dissertation consists of two parts:

1. fault-local self-healing, and

2. specification-based design of self-healing.

In the first part, we present our work on fault-containment to achieve scalable self-healing with respect to network size. In Chapter 2 we present a hierarchy-based fault-containment technique for fault-local self-healing of tracking, and in Chapter 3 we present a stretch-factor based fault-containment technique for fault-local self-healing of clustering. We discuss related work on fault-containment in Chapter 4.

In the second part, we present our work on specification-based design of self-healing to achieve scalability with respect to implementation size. To this end, we introduce two fault-tolerance preserving and compositional refinements, namely everywhere and convergence refinements, in Chapter 5. In Chapter 6, we illustrate the design of specification-based self-healing to our hierarchical tracking service presented in Chapter 2 in order to achieve scalability of design effort of self-healing with respect to the real world implementations of this tracking service. In Chapter 7, we discuss the soundness and completeness of the abstraction functions we use, and present a preliminary method for achieving automated synthesis of specification-based design of fault-tolerance. We present related work on design of fault-tolerance and fault-tolerance preserving refinements in Chapter 8.

Finally, we present concluding remarks in Chapter 9.

PART I
FAULT-LOCAL SELF-HEALING

CHAPTER 2

A HIERARCHY-BASED FAULT-LOCAL HEALING TECHNIQUE FOR TRACKING IN SENSOR NETWORKS

2.1 Introduction

Owing to applications in mobile computing, cellular telephony, and military contexts, tracking of mobile objects has received significant attention [16, 20, 38, 90, 100]. More recently, the DARPA Network Embedded Software Technology (NEST) program posed tracking as a challenge problem in wireless sensor networks, and several groups have delivered small-scale (using 100 node networks) tracking demonstrations: pursuer-evader tracking with 1 human controlled evader and 3 autonomous pursuers is showcased in [99], and detection, classification, and tracking of various intruders, such as persons and cars, are demonstrated in [7].

Besides the opportunities they provide for tracking of objects, wireless sensor networks also impose new challenges. Sensor nodes are energy constrained; algorithms that require an excessive communication burden are unacceptable as they drain battery power quickly. Sensor networks are fault-prone, message losses and corruptions and node failures are frequent; nodes can lose synchrony and programs can reach arbitrary states [59]. On-site maintenance is infeasible; sensor networks should be

self-healing. Moreover, self-healing should achieve fault-containment; otherwise a fault in one region of the network may contaminate the entire network and require a global correction, wasting the energy of the nodes and hindering the availability of the tracking service.

Contributions. Our contribution is to present a hierarchy-based fault-local stabilizing algorithm, namely STALK (Stabilizing TrAcking via Layered linKs), for tracking in sensor networks. Starting from an arbitrarily corrupted state, STALK satisfies its specification in time and work proportional to perturbation size (i.e., the size of the initially faulty area) instead of network size. This fault-local self-healing notion implies fault-containment: fault contamination is confined to an area proportional to the perturbation size. We achieve fault-containment by slowing propagation of information as the levels of the hierarchy underlying STALK increase, enabling the more recent information propagated by lower levels to override misinformation at higher levels.

Our scheme for achieving fault-containment does not interfere with the efficiency of tracking operations in the absence of faults. While achieving fault-local stabilization, STALK also adheres to the locality of tracking operations: a *find* invoked within distance d of the mobile object requires $O(d)$ time and communication cost (work) to reach the object, and a *move* of the object to distance d away requires $O(d * log(\text{network diameter}))$ time and work to update the tracking structure. Furthermore, STALK achieves seamless tracking of a continuously moving object by enabling concurrent executions of move and find operations.

Overview of STALK. For achieving scalability, STALK employs a hierarchical structure. For ensuring the locality of both find and move operations, STALK adopts a

partial information strategy. The tracking information is maintained with accuracy related to the distance from the mobile object: Nearby nodes that are relatively inexpensive to update have more recent and accurate information about the object, whereas far away nodes that are relatively expensive to update have older and more approximate information about the object.

Tracking structure. We assume a hierarchical partitioning of the sensor network into clusters based on radius. The tracking structure is a path rooted at the highest level of the hierarchy. Each process in the *tracking path* has at most one child, either at its level or one below it in the hierarchy, and the mobile object resides at the leaf of the tracking path, at the lowest level. Each process in the path points to a process that is generally closer to the object and has more recent information about its location.

Find operation. A find operation invoked at a process queries neighboring processes at increasingly higher levels of the clustering hierarchy until it encounters a process on the tracking path. Once the tracking path is found, the find operation follows it to its leaf to reach the mobile object.

Move operation. We implement move-triggered updates by means of two local actions, *grow* and *shrink*. The grow action enables a path to grow from the new location of the object to increasingly higher levels of the hierarchy and connect to the original path. The shrink action cleans branches deserted by the object. Shrinking also starts at the lowest level and climbs to increasingly higher levels. Despite that grow and shrink occur concurrently, we achieve the move operation successfully by using suitable values for the process timers, which actuates the execution of these actions.

Fault-local stabilization. We use two concepts for achieving fault-locality: hierarchical partitioning and level-based timeouts for execution of actions. The key idea is to wait for more time before updating a wider region's view. We employ larger timeouts when propagating an update to a higher level of the hierarchy and, thus, more recent updates coming from lower levels can catch-up to misinformed updates at higher levels. The latency imposed by waiting is a constant factor of the communication delay and does not affect the accessibility of the tracking structure.

A perturbation count for a given system state is the minimum number of processes whose state must change to achieve a consistent state of the system. For work and time calculations the level of "perturbed" processes are important; a fault hitting a level l process affects the entire level l cluster and hence its size is r^l. We define *perturbation size* to be a weighted sum of the levels of perturbed processes. A stabilizing system is *fault local stabilizing* if the time and work required for stabilization are bounded by functions of perturbation size rather than system size.

Concurrent move and find operations. STALK achieves seamless tracking of a continuously moving object: An object can relocate before the effects of its previous move operations finish updating the tracking path, and a find operation may be concurrently in progress with these move operations. During concurrent move operations, it is not possible to achieve a complete tracking path; there will be discontinuities in the path. By giving an upperbound on the speed of the object, we prove a reachability condition on the tracking path and ensure that if a find encounters a dead-end while following a path, there is always an available newer path nearby.

Related work. STALK provides a "network middleware support" for tracking: it assumes an underlying service for detection of a mobile object [56,75,111] and provides a basis for higher level applications such as multiple target tracking [96] and pursuer-evader applications [48].

The idea of employing a hierarchical structure for achieving scalability of tracking has been extensively researched [3, 110] in the context of personal communication systems and mobile Internet Protocol, and the idea of using a partial information strategy to optimize both finds and moves has been investigated in [16, 23].

In [16], a hierarchy of regional directories is constructed so that each level l directory enables a node to find a mobile object within 2^l distance from itself. The communication cost of a find for an object d away is $O(d * log^2 N)$ and that of a move of distance d is $O(d * logD * logN + log^2 D/logN)$ (where N is the number of nodes and D is network diameter). A topology change, such as a node failure, necessitates a global reset of the system since the regional directories depend on a non-local clustering program [15] that constructs a sparse cover of a graph. In [23], the tracking problem is considered for a geometric network model similar to ours, and cost complexity similar to ours is achieved.

STALK offers properties that these protocols lack, such as fault-tolerance and seamless tracking of a continuously moving object. STALK is not only fault-tolerant but also stabilizing and fault-containing as well: Starting from an arbitrarily corrupted state, STALK recovers within work and time proportional to the size of the faulty region. STALK achieves seamless tracking of continuously moving objects: An

[0]The move operation in [23] costs O(d * log d) work (where d is the distance moved by the object), but their interpretation of "amortized cost" is more permissive than ours. Using the same interpretation for "amortized cost", the move operation in STALK also costs O(d * log d) work.

object can relocate before the effects of its previous move operations finish updating the tracking path, and a find operation may be concurrently in progress with these move operations.

There has been some work on self-stabilizing tracking algorithms [33, 38, 51]. The distributed arrow protocol [51] suffers from the dithering problem —where an object moving back and forth across a multi-level hierarchy boundary may lead to nonlocal updates. The protocols in [33] do not exploit the hierarchy idea and are not scalable for large networks. In [38], using a hierarchy of location servers, a stabilizing location management protocol is presented. However, in contrast to STALK, the protocol in [38] fails to ensure locality of finds. Also, none of these protocols enjoy fault-containment.

The area of fault-containment of self-stabilizing algorithms has received growing interest [12,17,45,86]. The notion of fault containment within the context of stabilization is formalized first in [45]; algorithms were proposed to contain state-corruption of a single node in a stabilizing spanning tree protocol. In [86] fault-containment of Byzantine nodes have been studied for dining philosophers and graph coloring algorithms; this work requires the range of contamination to be constant and is too limiting for problems such as tracking and routing whose locality are not constant. In [17], a broadcast protocol is proposed to contain observable variables in the presence of state corruptions, but the protocol allows for global propagation of internal protocol variables. We present a more detailed survey of the fault-containment and fault-local stabilization literature in Chapter 4.

A protocol that achieves fault-local stabilization in shortest path routing is presented in [12]. To achieve fault-containment the protocol uses containment actions

that are a constant time faster than the fault-intolerant program actions. In contrast to [12], we do not have a privileged set of containment actions in STALK; the program actions serve to this end. We enable fault-containment in a hierarchy-based manner by suitably varying the speed of actions (through the use of process timers) as per the level of the hierarchy they are executed at.

Organization of the chapter. After presenting the model in the next section, we present specifications of STALK in Section 2.3. In Section 2.4, we present the move operation. Fault-local stabilization of the tracking path is discussed in Section 2.5. The find operation is in Section 2.6. In Section 2.7 we discuss concurrent execution of move operations, where the mobile object may relocate while previous move operations are still updating the tracking structure. In Section 2.8 we consider execution of find operations while moves are concurrently updating the tracking structure. Finally, we conclude the chapter in Section 2.9. We refer the readers to the technical report [35] for the detailed proofs.

2.2 Model

We consider a sensor network consisting of multiple sensor locations. Each sensor location plays host to (possibly) multiple processes with identifiers from a set P. In this chapter, as a convention, i and j refer to process identifiers, and $i.x$ refers to the value of variable x at i.

We denote the location of a process i with $loc(i)$ (and for convenience the set of locations of process set I with $loc(I)$). The Euclidean distance between the locations of i and j is denoted by $dist(i, j)$.

Hierarchical partitioning. Assume a hierarchical partitioning of processes over locations. Consider a tree with levels 0 through MAX of all processes P. For each process i we define:

1. $lvl(i)$, the level of process i in the tree,
2. $h(i)$, i's parent in the tree (for convenience, we define $h(i)$ to be i if $lvl(i) = MAX$),
3. $h^n(i)$, the iterated parent, defined as $h(i)$ if $n = 1$ and $h(h^{n-1}(i))$ otherwise,
4. $children(i)$, i's children in the tree.

We assume a one-to-one correspondence between the level 0 processes in the tree and node locations. For a location v we denote the level 0 process residing at v as $proc_0(v)$. We also assume that for any i such that $lvl(i) > 0$, i's location $loc(i)$ is equal to $loc(j)$ of one of its children j. This partitioning yields *clusters*. For i such that $lvl(i) = k+1,\ 0 \leq k < MAX$, $children(i)$ together form a cluster C at level k whose clusterhead is i. Radius of cluster C is the maximum distance from i to any process in C. Next we introduce the symmetric neighbor relation. For level 0 processes $i, j,\ i \neq j,\ \ j \in nbr(i) \iff dist(i, j) \leq 1$. For level $k > 0$ processes i, j, that are clusterheads of level $k-1$ clusters C_i and C_j, i and j are neighbors if C_i contains a process that is a neighbor of a process in C_j.

Geometry assumptions. We fix the following assumptions about the hierarchical partitioning:

1. We define a real constant $r \geq 3$ to denote the cluster dilation factor; the radius of a level l cluster is at least r^l,

2. We define a real maximum cluster radius constant $m \geq 2/\sqrt{3}$ to bound the radius of a level l cluster to be at most mr^l,

3. We define a real minimum cluster breadth constant q satisfying $\frac{2m+r-1}{r-1} \leq q \leq 2m$ to restrict the locations in *non-neighboring* level l clusters to be greater than qr^l apart.

The constraints imply a bound, ω, on the number of neighbors at any level $l > 0$. They also imply that, for $l > 0$, the distance between two neighboring level l processes is within $2r^{l-1}$-to-$2mr^{l-1}$, and the distance between a level l process and its children in the hierarchy is at most mr^{l-1}. This clustering does not necessarily imply a uniform tiling of the network, as radii of clusters at the same level are not required to be the same. The network diameter, D, is the maximum distance between any two locations in the network. Each node in the network is deployed with $O(MAX)$ storage where $MAX \leq log_r D$.

An example of the clustering geometry with $r = 3$ can be found in Section 2.4. Our hierarchical partitioning constraints can be realized by using a distributed and fault-local stabilizing clustering protocol, FLOC [34].

2.3 System Specification

Here we describe the specification for STALK modeled as I/O automata.

Mobile object. The mobile object **Evader** resides at exactly one sensor location. An $\mathbf{object}_i$ occurs at all processes residing at the object's current location and $\mathbf{no\_object}_j$ occurs for all other locations. When moving, the object nondeterministically moves to a neighboring location.

STALK. STALK consists of two parts, **Tracker** and **Finder**, as seen in Figure 2.1. **Tracker** maintains a tracking structure by propagating mobile object information obtained through **object** and **no_object** inputs. **Finder** answers client **find**s by outputting **found** at the mobile object's current location. **Finder** queries **Tracker** for location information through **cpq** requests and **Tracker** answers with **cpointer** responses.

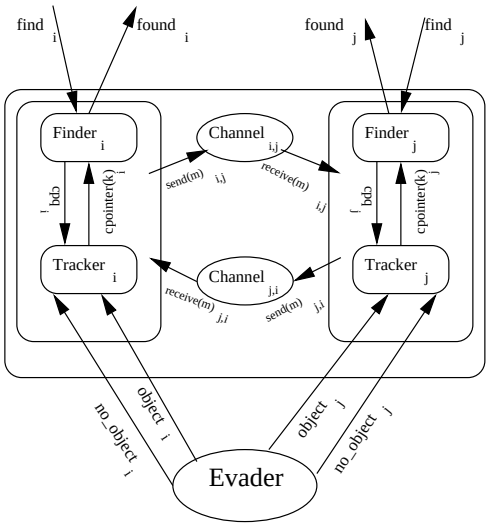


Figure 2.1: STALK architecture.

Both parts are implemented distributedly by individual processes communicating through channels. Each process is assumed to have access to its own local timer, that advances at the same rate at all processes. We do not assume time synchronization across processes.

Channels. We use a communication abstraction of a (possibly) multi-hop channel $\mathbf{Channel}_{i,j}$ between any two processes i and j. Such channels are accessed using $\mathbf{send(m)}_{i,j}$ to send from i and $\mathbf{receive(m)}_{i,j}$ to receive at j. The cost of sending a message through $\mathbf{Channel}_{i,j}$ is $dist(i,j)$, and in the absence of faults a message is removed from the channel by at most $\delta * dist(i,j)$ time where δ is a known message delay factor.

Fault model and tolerance specification. Processes can suffer from arbitrary state corruption. These faults may occur at any time and in any finite number and order. Channels may suffer faults that corrupt, manufacture, duplicate, or lose messages.

We say a system is *self-stabilizing* iff starting from an arbitrary state the system eventually recovers to a consistent state, a state from where its specification is satisfied. In Section 2.4 we characterize consistent states for our implementation.

A perturbation count for a given system state is the minimum number of processes whose state must change to achieve a consistent state of the system. For work and time calculations the level of "perturbed" processes are important; a fault hitting a level l process affects the entire level l cluster and hence its size is r^l. We define

perturbation size to be a weighted sum of the levels of perturbed processes. A stabilizing system is *fault local stabilizing* if the time and work required for stabilization are bounded by functions of perturbation size rather than system size.

Complete system. The complete system is the composition of all channels, **Evader** and STALK.

We require the system be fault-local stabilizing to a consistent state. Starting from a consistent state with no outstanding **find** requests and no process or channel corruptions, we require that:

1. A **find** is eventually followed by a **found** at a location hosting the mobile object,
2. Each **found** is in response to a prior unanswered **find**,
3. If a **find** is initiated at a process Euclidean distance d from the mobile object, the time and work (communication) performed to service it is at most $O(d)$,
4. If the object moves d distance, the amortized time and work to update the tracking structure is $O(d * log(D))$.

2.4 Tracker

Here we describe how **Tracker** updates the tracking path after a move, assuming that the mobile object does not relocate until the updates are completed. In Section 2.7, we relax this restriction and allow the object to relocate while effects of its previous moves are still rippling through the path.

Updates to the tracking path are implemented by two local actions, grow and shrink. The grow action enables a new path to grow to increasingly higher levels of the clustering hierarchy and connect to the original path at some level. The shrink

action cleans old branches deserted by the mobile object starting from the lowest levels and climbing to increasingly higher levels. We present the grow action in Section 2.4.1 and the shrink action in Section 2.4.2.

A hierarchical partitioning of a network inevitably results in multi-level cluster boundaries: even though two processes are neighbors they might be contained in different clusters at all levels (except the top) of the hierarchy. If a process were to always propagate grows and shrinks to its clusterhead, a small movement of the object back and forth across a multi-level cluster boundary could result in work proportional to the size of the network rather than the distance of the move. To resolve this "dithering" problem, we allow one *lateral link* per level in our tracking path. A process occasionally connects to the original path with a lateral link to a neighboring process rather than by propagating a link to its parent in the hierarchy. We limit the lateral link count per level in order not to upset the locality properties of the find operation.

To implement **Tracker**, each process i maintains a child pointer c, a parent pointer p, a grow timer $gtime$, and a shrink timer $stime$. In the initial states, $i.c = i.p = \perp$ and $i.gtime = i.stime = \infty$ for all i. We assume the use of grow and shrink constants g and s that satisfy:

$$s \geq 10.5\delta m \tag{2.1}$$

$$\frac{s + \delta m}{r} < g \leq s - \delta m \tag{2.2}$$

A grow or shrink timer is set at i for $g * r^{lvl(i)}$ or $s * r^{lvl(i)}$ time respectively. The values for the timers are chosen to satisfy the requirements on both the work calculations in Section 2.4.4 and the fault-containment proofs in Section 2.5.

Tracker$_i$ has four inputs: **object**$_i$, **no_object**$_i$, **cpq**$_i$, and **receive**$(msg)_{j,i}$, and two outputs: **cpointer**$(j)_i$ and **send**$(msg)_{i,j}$ (msg can be gquery, ack_gquery, grow, or shrink).

Tracker$_i$ answers a **cpq**$_i$ input (an information request from **Finder**$_i$) with a **cpointer**$(i.c)_i$ output, providing the value of its child pointer. The **send**s and **receive**s propagate grows and shrinks as explained in detail below for process i.

2.4.1 Grow action

A grow updates a path to point to the new location of the object.

If i is at level 0, the object is at the same location as i, and i's child pointer c does not point to itself, then i becomes the leaf of the tracking path by setting c to i and setting its grow timer, *gtime*, scheduling a **grow** to be sent when *gtime* expires.

If i is above level 0 and receives a **grow** message, it sets its c pointer to the sender, sets *gtime* scheduling a **grow** to be sent to its prospective parent. i also sends a **gquery** message to its neighbors to check if the tracking path is reachable through a neighbor. The tracking path allows the use of one lateral link per level. A neighbor j that receives the **gquery** sends an **ack_gquery** back if j is on the tracking path and there isn't already a lateral link pointing to j, i.e., if $j.p$ points to its own clusterhead, $h(j)$. If i receives such an **ack_gquery** from j then it sets p to point to j, in preparation for adding a lateral link at j.

When *gtime* expires, if c is still non-$\perp$, meaning that the path has not shrunk while i's grow timer was counting down, then a **send (grow)** is performed to extend the tracking path. If $i.p$ points to a neighbor j then the grow message is sent to j, inserting a lateral link. Otherwise, if $p = \perp$, i sets p to point to its own clusterhead

$h(i)$ and sends a **grow** message to $h(i)$, propagating the grow one level up in the hierarchy. In either case *gtime* is set to ∞, and i's role in updating the tracking path is complete.

If a **grow** message is received at i but i already has a parent in the tracking path or is the MAX level process, then i does not propagate the grow (it is already on the tracking path).

The grow actions at process i are in Figure 2.2.

2.4.2 Shrink action

A shrink cleans old, deserted branches of the tracking path.

If i is at level 0 and has a non-$\perp$ child pointer, but the mobile object is not at i's location, then i removes itself from the leaf of the tracking path. It sets its child pointer c to $\perp$ and sets the shrink timer *stime*, scheduling a **shrink** to be sent upon expiration of *stime*.

If i receives a **shrink** message from another process j, i checks to see whether its child pointer c points to j (c might not point to j; it may have been updated to point to a process on a newer path). If $c = j$ then i removes itself from the path by setting c to $\perp$ and then sets its shrink timer, scheduling a **shrink** message to be sent to its parent p. Otherwise, if $c \neq j$, i ignores the message, ensuring that shrink actions clean only deadwood and not the entire tracking path.

When *stime* expires, if c is still $\perp$, meaning no newer path has connected at i while *stime* was counting down, i sends a **shrink** message to its parent p in the path and then sets p to $\perp$.

Input: $\textbf{object}_i$
eff: if $c \neq i \ \wedge \ lvl(i) = 0$ then
$c := i$
$gtime := now + g$

Output: $\textbf{send (gquery)}_{i,j}$
pre: $j \in gnbrquery$
eff: $gnbrquery := gnbrquery - \{j\}$
if $gnbrquery = \emptyset$ then
$gtime := now + g * r^{lvl(i)}$

Input: $\textbf{receive (gquery)}_{j,i}$
eff: if $p = h(i)$ then
$gqack := j$

Output: $\textbf{send (ack\_gquery)}_{i,j}$
pre: $gqack = j$
eff: $gqack := \perp$

Input: $\textbf{receive (ack\_gquery)}_{j,i}$
eff: if $c \neq \perp \ \wedge \ p = \perp$ then
$p := j$

Output: $\textbf{send (grow)}_{i,j}$
pre: $now = gtime \ \wedge \ c \neq \perp \ \wedge$
$((j = p \ \wedge \ p \in nbr(i)) \ \vee \ (j = h(i) \ \wedge \ p = \perp))$
eff: if $p = \perp$ then
$p := h(i)$
$gtime := \infty$

Input: $\textbf{receive (grow)}_{j,i}$
eff: $c := j$
if $lvl(i) = MAX$ then
$p := i$
if $p = \perp$ then
$gnbrquery := nbr(i)$

Figure 2.2: Grow actions at process i

Input: **no_object**$_i$
eff: if $lvl(i) = 0 \ \wedge \ c \neq \perp$ then
$c := \perp$
$stime := now + s$

Output: **send (shrink)**$_{i,j}$
pre: $now = stime \ \wedge \ c = \perp \ \wedge \ j = p$
eff: $p := \perp$
$stime := \infty$

Input: **receive (shrink)**$_{j,i}$
eff: if $c = j$ then
$c := \perp$
$stime := now + s * r^{lvl(i)}$

Figure 2.3: Shrink actions at process i

The shrink actions for process i are in Figure 2.3.

Example. Figure 2.4 depicts a sample tracking path. The path is seen pointing to a level 2 clusterhead, which points to one of its hierarchy children, a level 1 clusterhead. That clusterhead has a lateral link to another level 1 clusterhead that points to a level 0 cluster where the object **e** is located. Deadwood is denoted by the dotted path.

2.4.3 Correctness

Here we present system invariants and define consistent states of the system.

In the absence of faults, every process i satisfies I, the following five conditions, at all times:

$I0$. If $lvl(i) = 0$ and **object**$_i$ occurs then $i.c = i$,

$I1$. If $i.c \neq \perp$ then one of the following holds:

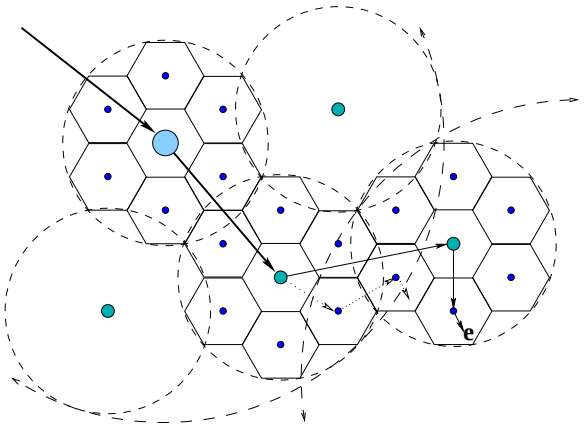


Figure 2.4: Tracking path example

(a) $i.c = i$ and the object is at i,

(b) $i.c$ points to one of its children in the clustering hierarchy, or

(c) $i.c$ points to a neighbor and $i.p$ points to its parent in the clustering hierarchy,

$I2$. If $i.p \neq \perp$ then either $i.c \neq \perp$ or i is executing a shrink action and will send a **shrink** to $i.p$,

$I3$. The dual: if $i.c \neq \perp$ then $i.p \neq \perp$ or i is executing a grow action and will send a **grow** to its prospective parent,

$I4$. If $i.c \neq i$ and $i.c \neq \perp$ then $(i.c).p$ is either i or $\perp$. In the latter case a **shrink** from $i.c$ is in transit to i.

A *tracking path* is a sequence $\{i_x, \ldots, i_1\}$ where i_1 is a leaf and contains the object, every process but i_1 points to the next process as its child, and I is satisfied at all processes in the sequence. A *complete tracking path* is a tracking path $\{i_x, \ldots, i_1\}$ where $lvl(i_x) = MAX$ and $i_x.p = i_x$.

A *consistent state* is a state where a complete tracking path exists and $i.c = i.p = \perp$ for every process i not in the tracking path. Using invariant I it follows from the program actions that an execution starting from an initial state eventually reaches a consistent state and that consistent states are closed under moves of the object.

2.4.4 Work

In order to prove our work claims, we must show that the timing of changes to the new and old tracking paths satisfy certain relationships to ensure that the old path is reused (via insertion of a lateral link) to the extent possible. More specifically, it follows from the assumptions on timer constants s and g that an old path being cleaned bottom-up from level 0 will not clean one of its level l pointers before a grow starting at level 0 in the new path reaches level l and has an opportunity to query one of those pointers, allowing for the addition of a lateral link.

This allows us to reason that the new path (which grows by propagating pointers straight up the hierarchy until it connects to the old path) connects to the pre-shrink old path at the lowest level process that is either an iterated clusterhead of the new object location or a neighbor of such a clusterhead that is not itself connected to the tracking path via a lateral link. In the latter case, the new path would connect via a lateral link.

We then prove the following theorem.

Theorem 2.1 *Starting from a consistent state, move operations of the mobile object to a total of distance d away require at most $O(d * \omega mr * MAX)$ amortized work and $O(d * gr^2 * MAX)$ amortized time to update the tracking path.*

Proof sketch. The above reasoning implies a level l pointer in the path is updated as often as every $\sum_{j=1}^{l-2} qr^j$ distance because of the required use of lateral links at all levels below l (note that qr^l is the minimum distance between two non-neighboring level l clusters). An $O(mr^{l-1})$ work and $O(gr^l)$ time cost is incurred each time a level l pointer is updated. The costs, multiplied by frequency of updates, are summed for each level for the result. □

2.5 Fault-Containment

After state corruption of a region of (potentially all) processes, our tracking path heals itself in a fault-local manner within work proportional to perturbation size. Here we discuss correction actions enabling fault-local stabilization of the path.

Through faults a shrink action can be mistakenly initiated. For example, when a portion of a tracking path is hit by faults, higher level processes of the path, unaware a healthy lower path exists, start a shrink action. If "growth" at lower levels lags behind "shrinking" of upper levels, faults can propagate through the entire upper path. For fault-containment, grow actions started at lower levels must contain shrink actions.

Similarly, grow actions can be mistakenly initiated. Consider a garbage path with no object at its leaf. The topmost process of this path, unaware that the path does not lead to the object, starts a grow action. If "shrinking" from lower levels lags behind "growing" of upper levels, faults can contaminate the entire network. Thus shrinks started at lower levels must contain grows.

The above requirements are both satisfied by giving priority to actions with more recent information regarding the path; actions from lower levels are privileged over

ones at higher levels. We achieve this by delaying shrink/grow for longer periods as the level of the process executing the action increases. This way, propagation actions coming from below are subject to lesser delays and can arrest mistakenly initiated propagation actions; fault-local stabilization is achieved. We note that the latency imposed by delaying is a constant factor of the communication delay to higher levels and does not affect the quality of tracking.

Stabilization. Here we present correction actions for re-establishing the tracking path invariant I starting from an arbitrarily corrupted state.

Correction of $I0$ and $I1$. $I0$ is established trivially by **object** and **no_object** inputs. The correction of $I1$ follows from the domain assumptions we make on non-$\perp$ c, p and $gnbrquery$ variables for $i \in P$. We require that $i.c \neq \perp \Rightarrow i.c \in \{nbr(i) \cup children(i)\}$: $i.c$ points to either a neighbor of i or to a child of i. Similarly, we restrict the domain of non-$\perp$ $i.p$ variables to $\{nbr(i) \cup \{h(i)\}\}$ and $i.gnbrquery$ to subsets of $nbr(i)$. These assumptions are reasonable since the clustering provides a process with the identifiers of its neighbors, children, and clusterhead; a process can locally check and set these variables to $\perp$ if their values are outside their respective domains.

Correction action for $I2$. If i has a valid parent but no valid child, then $I2$ is corrected at i by setting $i.c = \perp$ and scheduling a **shrink** message to be sent to $i.p$.

Correction action for $I3$. If i has a valid child but no parent, then a **gquery** message is sent to i's neighbors and a **grow** message is scheduled to be sent to the future parent of i.

The correction actions for $I2$ and $I3$ are given in Figure 2.5.

Internal: **start-shrink**$_i$
pre: $(c = \perp \wedge p \neq \perp \wedge stime \notin [now, now + s * r^{lvl(i)}])$
$\vee [p \in nbr(i) \ \wedge \ c \in nbr(i)]$
eff: $c := \perp$
$stime := now + s * r^{lvl(i)}$

Internal: **start-grow**$_i$
pre: $c \neq \perp \ \wedge \ p = \perp \ \wedge \ gtime \notin [now, now + g * r^{lvl(i)}]$
eff: if $lvl(i) = MAX$ then
$p = i$
if $p = \perp$ then
$gnbrquery := nbr(i)$

Figure 2.5: Starting grow/shrink at process i

Correction actions for I4. To correct $I4$ we use heartbeat messages and two timers: *next* for periodically sending heartbeats to the parent and a *timeout* for dissociating a child if no heartbeat is heard. The correction actions use a constant b for calculating the frequency of heartbeat messages, whose periodicity are tunable to achieve less communication or faster detection. We require that b is more than twice s, the shrink timer constant:

$$b \geq 2s \tag{2.3}$$

Intuitively, this condition serves to prevent a scenario where aggressively scheduled heartbeats shrink the original path before a new growing path can reconnect to the original.

Every i with a non-$\perp$ valued parent sends a **heartbeat** message to its parent every $b * r^{lvl(i)}$ time by setting *next*. Every time i receives a **heartbeat** or **grow** message from its child, *i.e*, i resets its *timeout* variable to $(b + 2\delta m/r) * r^{lvl(i)}$ (it is

also reset upon receipt of a **grow** to prevent the scenario where the heartbeat timeout of i expires scheduling a shrink just after i receives a **grow** message from a process in a newly growing path). If i receives a heartbeat from j but $i.c = \perp$ then i sets $i.c := j$. Otherwise, a **heartbeat** message received from a process other than $i.c$ is ignored.

If i has a non-$\perp$ valued child, is not a leaf, and has not received a **heartbeat** message in a $(b + 2\delta m/r) * r^{lvl(i)}$ time interval, then $i.c$ is set to $\perp$.

The correction actions at i for $I4$ are in Figure 2.6.

Output: **send (heartbeat)**$_{i,j}$
pre: $now = next \ \wedge \ j = p$
eff: $next := now + b * r^{lvl(i)}$

Input: **receive (heartbeat)**$_{j,i}$
eff: if $c = \perp$ then $c := j$
if $c = j$ then
$timeout := now + (b + 2\delta m/r) * r^{lvl(i)}$

Internal: **timeout_expire**$_i$
pre: $now = timeout \ \wedge \ c \neq \perp \ \wedge \ c \neq i$
eff: $c := \perp$

Internal: **heartbeat_set**$_i$
pre: $p \neq \perp \ \wedge \ next \notin [now, now + b * r^{lvl(i)}]$
eff: $next := now + b * r^{lvl(i)}$

Internal: **timeout_set**$_i$
pre: $c \neq \perp \ \wedge \ c \neq i$
$\wedge \ timeout \notin [now, now + (b + 2\delta m/r) * r^{lvl(i)}]$
eff: $timeout := now + (b + 2\delta m) * r^{lvl(i)}$

Figure 2.6: Heartbeat actions at process i

Stabilization of the *next* and *timeout* variables of the corrector is ensured by keeping their values within their respective domains. Using the correction actions described above, we prove in Theorem 2.2, that STALK is self-stabilizing to a consistent state, where a complete tracking path exists.

Theorem 2.2 STALK *is self-stabilizing.* □

Fault-local stabilization. To prove fault-local stabilization we first give a bound on arresting distance of grow/shrink actions in Lemmas 2.3 and 2.4. In these lemmas, $l_1 + 1$ and l_2 are respectively the lowest and highest perturbed levels: faults occur only from level $l_1 + 1$ through level l_2. We prove fault containment by showing that due to our timing assumptions, a correction propagated from l_1 catches propagation of bad information at a level $l > l_2$, leaving levels above l untouched by faults. The proof is done by comparing the maximum time the propagation of a lower wave takes to reach l versus the minimum time the higher wave takes to pass it.

Lemma 2.3 *Propagation of a shrink action started at level* $l_1{+}1$ *catches propagation of a grow action started at level* l_2 *by level* l *where*

$$l = l_2 + \lceil \log_r \tfrac{br-b+sr+gr-2s+3\delta m}{gr-s-\delta m} \rceil.$$ □

Lemma 2.4 *Propagation of a grow action started at level* l_1 *catches propagation of a shrink action started at level* l_2 *by level* l *where*

$$l = l_2 + \lceil \log_r \tfrac{br-b+sr^2-gr-\delta m}{sr-gr-3\delta m} \rceil.$$ □

The size, $l - l_2$, of contamination due to fault propagation is independent of the network size and is tunable via grow and shrink timer settings. In Section 2.7 we give sample values for these.

Theorem 2.5 (Fault-local stabilization) *For a perturbation size S, our program self-stabilizes in $O(S)$ work and in $O(r^L)$ time where L denotes the highest perturbed level.* □

The proof for Theorem 2.5 follows from the Lemmas 2.3 and 2.4.

2.6 Finder

Here we describe **Finder** assuming find operations are interleaved with move operations. We relax this restriction in Section 2.8 and allow the object to relocate while a find is in progress.

A find consists of two phases: *searching* and *tracing*. Searching queries neighboring processes at increasingly higher levels of the hierarchy until a tracking path is found. Tracing then follows the pointers in the tracking path to the mobile object.

A client initiates the operation with a **find** input. The level 0 process at that location starts servicing the find.

A find is serviced at a process i by first querying the local $\mathbf{Tracker}_i$ using $\mathbf{cpq}_i$. $\mathbf{Tracker}_i$ will then return its child pointer c' through $\mathbf{cpointer}(c')_i$.

If $c' = i$, the object is found at i and the tracing phase is over, so i outputs $\mathbf{found}_i$.

If $c' \neq i$ and $c' \neq \perp$, the tracing phase is continuing, and i sends a **find** to process c'.

If $c' = \perp$ it is still the search phase, and i sends an **fquery** message to its neighbors and sets a timeout equal to the maximum time for roundtrip neighbor communication at $lvl(i)$. Neighbors answer the query with an **fqack** message and start servicing the find if they are on the tracking path, and ignore it otherwise. If such an **fqack** is received before the timeout period expires at i, i knows the tracking path has been

found and tracing has started at j; i is done. If the timeout period expires with no reply from a neighbor, the search phase is continuing. In this case i sends a **find** to its clusterhead and hands over the responsibility for servicing the find to $h(i)$.

Work. Finds are local: a find initiated at process i distance d from the mobile object requires $O(d)$ work to complete. To see this we first note that geometry assumptions imply:

Theorem 2.6 (Proximity) *In a consistent state, for a process j that is at most d distance from the mobile object, one of the following holds:*

- $h^{\lceil log_r d \rceil + 1}(j)$ *is in the tracking path or*
- $\exists i \in nbr(h^{\lceil log_r d \rceil + 1}(j))$ *in the tracking path.*

Proof sketch. Say $d = r^l$. For this theorem to be false, it must be that level l cluster j is represented by does not neighbor any level l cluster in the tracking path, implying the distance between j and any process represented by a level l cluster on the tracking path is more than qr^l. However, using the fact that there is at most 1 lateral link per level and that the maximum radius of a level l cluster is mr^l, we can conclude that the distance between the leaf of the tracking path and any process represented by a level l cluster in the tracking path is at most $\sum_{j=0}^{l-1} 2mr^j$. This plus the distance r^l is less than qr^l, by assumptions in Section 2. □

Theorem 2.7 *A* find *operation invoked at distance d from a mobile object results in $O(d * \omega rm)$ work and takes $O(d * \delta rm)$ time.*

Proof sketch. The previous theorem implies a find operation will find the path by level $\lceil log_r d \rceil + 1$. We add this cost of searching to the cost of following tracking path links from that level. □

2.7 Concurrent Move Operations

In this section we relax the atomic move restriction and consider concurrent execution of move operations, where the mobile object may relocate while effects of previous move operations are still rippling through the tracking structure.

We showed that a complete tracking path was preserved by atomic move operations. However, during concurrent move operations, we can not guarantee a complete tracking path: at any given instant, there may be a new path growing, older deadwood shrinking, and new deadwood being produced. Hence, we provide a looser definition of a *tracking structure* consisting of several path segments that satisfy a reachability condition.

A *path segment* is a piece of a tracking path. The piece is maximal in that the first process in the segment has no parent pointer or has a parent pointer to itself (for the topmost level of hierarchy) and the last pointer in the segment (the endpoint) points to itself or a process without a pointer. A sequence of path segments $\{\{i_{x,y_x}, \ldots, i_{x,1}\}, \ldots, \{i_{1,y_1}, \ldots, i_{1,1}\}\}$ is a *tracking structure* if $i_{1,1}$ contains the object and every endpoint i, s.t. $i = i_{y,1}, y \neq 1$, satisfies a 3-part *reachability condition*: *(1)* If $i.c$ is i's hierarchy child, $lvl(i) > 1$ implies the next path segment contains a neighbor of i, and $lvl(i) = 1$ implies the next segment contains a process neighboring $i.c$. *(2)* If $i.c$ is i's neighbor, the next segment contains a process neighboring $i.c$. *(3)* If $lvl(i) > 1$, the next segment's endpoint is at least 2 levels below $lvl(i)$.

A *complete tracking structure* is a tracking structure that reaches the top level of the hierarchy. We also define a weaker version of a consistent state: A *good state* is a program state where a complete tracking structure exists and $i.c = i.p = \perp$ for all processes i not in the tracking structure.

We assume the object takes at least e time at a level 0 process (that is, within the coverage area of its sensor) before moving to a neighboring level 0 process, and the minimum time the object takes to move a total of d distance is $e * d$ where

$$e \geq 2sr^3 \tag{2.4}$$

Theorem 2.8 *Starting from a good state, a move of the object leads to another good state.*

Proof sketch. The reachability condition is implied because the time a mobile object takes moving far enough to require a level $l-2$ update and then propagating a shrink to remove the level $l-2$ pointers is more than the time to delete level l pointers in a prior segment. □

The following two theorems have proofs very similar to those of the non-concurrent case.

Theorem 2.9 *Starting from a good state, object moves to distance d away take $O(d* \omega rm * MAX)$ work and $O(d * gr^2 * MAX)$ time to complete.*

Proof sketch. Newer segments do not outgrow older segments so Theorem 2.1 still holds. □

Theorem 2.10 (Fault-local stabilization) *For concurrent moves and perturbation size S the system self-stabilizes to a good state in $O(S)$ work and in $O(r^L)$ time where L denotes the highest perturbed level.* □

Sample timer constants. Consider $g = 5\delta m, s = 11\delta m, e = 23\delta mr^3$, and $b = 11\delta mr$. Tracking structure inequalities are satisfied, grow actions catch faulty shrink actions in 2 levels, and shrinks catch faulty grows within 4 levels.

2.8 Concurrent Find and Move Operations

Given our prior timing assumptions, find operations are successful even when move operations are still in progress on the tracking structure.

In the searching phase a find invoked within d distance of a mobile object hits the tracking structure by level $\lceil log_r d \rceil + 1$ as before; the object can not move fast enough to result in a propagation of a shrink to level $\lceil log_r d \rceil + 1$ before the find operation gets there.

In a tracing phase that is concurrent with a move, a complete tracking path may not be available, and a find may reach a process with $c = \perp$ while tracing the tracking structure. If a find reaches such a dead end it re-executes the searching phase. The reachability condition of the tracking structure ensures the find will reach a newer path segment by searching neighboring processes at the current level or one level higher. The mobility of the object only results in a constant factor difference in time and work to complete a find.

Theorem 2.11 *A find operation invoked within d distance of a mobile object requires $O(d\omega rm)$ work and $O(d\delta rm)$ time to reach the object.* □

2.9 Chapter Summary

We presented STALK, a fault-local stabilizing tracking service for sensor networks. We use two concepts to achieve fault locality: hierarchical partitioning and level-based timeouts for execution of actions. The key idea is to wait longer before updating a wider region's view by employing larger timeouts when propagating an update to higher levels of the hierarchy. This way, more recent updates from lower levels can

catch-up to and override the misinformed updates at higher levels. While achieving fault-local stabilization STALK also adheres to the locality of tracking operations. Moreover, by enabling concurrent move and concurrent find operations STALK achieves seamless and continuous tracking of the mobile object.

In this work we focused on the analytical worst-case performance of STALK. In the appendix, we provide simulations for the average-case performance of STALK by considering a random movement model for the object. Simulation results show that work for move scales better than linearly due to the locality of movements featured by the random model. The code is available at `www.cse.ohio-state.edu/~demirbas/track/`.

STALK has applications in message routing to mobile units and in pursuer/evader games. As part of our efforts to develop sensor network services in the DARPA/NEST program, we are implementing STALK on the Mica mote platform [54]. For future work, we are examining other problems that could benefit from our hierarchy-based local stabilization technique.

CHAPTER 3

A STRETCH-FACTOR BASED FAULT-LOCAL HEALING TECHNIQUE FOR CLUSTERING IN SENSOR NETWORKS

3.1 Introduction

Large-scale ad hoc wireless sensor networks introduce challenges for self-configuration and maintenance. Centralized solutions that rely on pre-defined configurer or maintainer nodes are unsuitable: Requiring all the nodes in a large-scale network to communicate their data to a centralized base-station depletes the energy of the nodes quickly due to the long-distance and multi-hop nature of the communication and also results in network contention.

Clustering is a standard approach for achieving efficient and scalable control in these networks. Clustering facilitates the distribution of control over the network. Clustering saves energy and reduces network contention by enabling locality of communication: nodes communicate their data over shorter distances to their respective clusterheads. The clusterheads aggregate these data into a smaller set of meaningful information. Not all nodes, but only the clusterheads need to communicate far distances to the base station.

To enable efficient and scalable control of the network, a clustering service should combine several properties. The service should achieve clustering in a fast and local manner: cluster formation and changes/failures in one part of the network should be insulated from other parts. Furthermore, the service should produce approximately equal-sized clusters with minimum overlap among clusters. Equal-sized clusters is a desirable property because it enables an even distribution of control (e.g., data processing, aggregation, storage load) over clusterheads; no clusterhead is over-burdened or under-utilized. Minimum overlap among clusters is desirable for energy efficiency because a node that participates in multiple clusters consumes more energy by having to transmit to multiple clusterheads.

In this paper we are interested in a stronger property, namely a solid-disc clustering property, that implies minimization of overlap. The solid-disc property requires that all nodes that are within a unit distance of a clusterhead belong only to its cluster. In another words, all clusters have a nonoverlapping unit radius solid-disc.

Solid-disc clustering is desirable since it reduces the intra-cluster signal contention: The clusterhead is shielded at all sides with nodes that belong to only its cluster, so the clusterhead receives messages from only those nodes that are in its cluster, and does not have to endure receiving messages from nodes that are not in its cluster. Solid-disc clustering also results in a guaranteed upper bound on the number of clusters: In the context of hierarchical clustering, minimizing the number of clusters at a level leads to lower-cost clustering at the next level. Finally solid-discs yield better spatial coverage with clusters: Aggregation at the clusterhead is more meaningful since clusterhead is at the median of the cluster and receives readings from all directions of the solid disc (i.e., is not biased to only one direction).

Equi-radius solid-disc clustering with bounded overlaps is, however, not achievable in a distributed and local manner. We illustrate this observation with an example for a 1-D network (for the sake of simplicity).

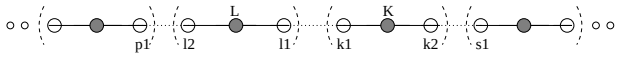


Figure 3.1: Each pair of brackets constitutes one cluster of unit radius, and colored nodes denote clusterheads.

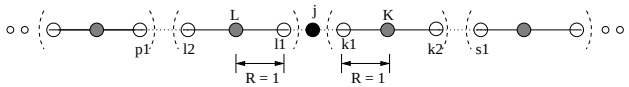


Figure 3.2: A new node j joins the network between clusters of clusterheads L and K.

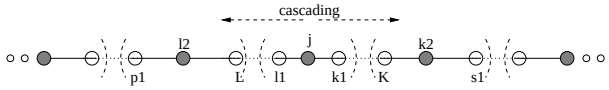


Figure 3.3: Node j forms a new cluster and leads to re-clustering of the entire network.

Consider a clustering scheme that constructs clusters with a fixed radius, say $R = 1$, solid-disc. Figure 3.1 shows one such construction. We show that for fixed radius clustering schemes, a node join can lead to re-clustering of the entire network. When node j joins the network (Figure 3.2), it cannot be subsumed in its neighboring clusters as j is not within unit distance of neighboring clusterheads L and K. j thus forms a new cluster with itself as the clusterhead. Since all nodes within unit radius of a clusterhead should belong its cluster, j subsumes neighboring nodes l_1 and k_1

in its cluster. This leads to neighboring clusterheads L and K to relinquish their clusters and election of l_2 and k_2 as the new clusterheads (Figure 3.3). The cascading effect propagates further as the new clusterheads l_2 and k_2 subsume their neighboring nodes leading to re-clustering of the entire network.

Our contributions. We show that solid-disc clustering with bounded overlaps is achievable in a distributed and local manner for approximately equal radii (instead of exactly equal-radii). More specifically, we present FLOC, a fast local clustering service that produces nonoverlapping and approximately equal-sized clusters. The resultant clusters have at least a unit radius solid-disc around the clusterheads, but they may also include nodes that are up to m, where $m \geq 2$, units away from their respective clusterheads. By asserting $m \geq 2$, FLOC achieves locality: effects of cluster formation and faults/changes at any part of the network are contained within at most m unit distance.

While presenting FLOC we take unit radius to be the reliable communication radius of a node and m to be the maximum communication radius. In so doing we exploit the double-band nature of wireless radio-model and present a communication- and, hence, energy-efficient clustering.

FLOC is suitable for clustering large-scale wireless sensor networks since it is fast and scalable. FLOC achieves clustering in O(1) time regardless of the size of the network. FLOC is also locally self-healing in that after faults stop occurring, faults and changes are contained within the respective cluster or within the immediate neighboring clusters, and FLOC achieves re-clustering within constant time.

We simulate FLOC using Prowler [97] and analyze the tradeoffs between clustering time and the quality of the clustering. We observe that forcing a very short clustering

time leads to network traffic congestion and message losses, and hence, degrades the quality of the resultant clustering. We suggest suitable parameters for FLOC to achieve a fast completion time without compromising from the quality of clustering. Furthermore, we implement FLOC on the Mica2 [102] mote platform and experiment with actual deployments to corroborate our simulation results.

Outline. After presenting the network and fault model in the next section, we present the basic FLOC program in Section 3.3. We discuss the self-healing properties of FLOC in Section 3.4. In Section 3.5, we present additional actions that improves the convergence time of the clustering. We discuss our simulation and implementation results in Section 3.6. In Section 3.7 we present related work, and we conclude the paper in Section 3.8.

3.2 Model

We consider a wireless sensor network where nodes lie in a 2-D coordinate plane. The wireless radio-model for the nodes is double-band: A node can communicate reliably with the nodes that are in its inner-band (*i-band*) range, and unreliably (i.e., only a percentage of messages go through) with the nodes in its outer-band (*o-band*) range. This double-band behavior of the wireless radio is observed in [25, 109, 112]

We define the unit distance to be the i-band radius. We require that the o-band radius is m units where $m \geq 2$. This is a reasonable assumption for o-band radius [25, 109, 112]. Nodes can determine whether they fall within i-band or o-band of a certain node by using any of the following methods:

- Nodes are capable of measuring the signal strength of a received message [54]. This measurement may be used as an indication of distance from the sender.

E.g., assuming a signal strength loss formula ($\frac{1}{1+d^2}$), where d denotes distance from the sender, the i-band neighbors receive the message with [0.5, 1] of the transmission power, and, for $m = 2$ the o-band neighbors receive the message with [0.2, 0.5] power.

- Nodes may maintain a record of percentage of received messages with respect to neighbors [25], and infer the i-band/o-band neighbors from the quality of the connections.
- An underlying localization service [82, 87] may provide the nodes with these distance information.

We assume that nodes have timers, but we do not require time synchronization across the nodes. Timers are used for tasks such as sending of periodic heartbeats and timing out of a node when waiting on a condition. Nodes have unique *id*s. We use i, j and k to denote the nodes, and $j.var$ to denote a program variable residing at j. We denote a message broadcast by j as msg_j.

A *program* consists of a set of variables and actions at each node. Each action has the form: $<guard> \longrightarrow <assignment\ statement>$. A $guard$ is a boolean expression over variables. An assignment statement updates one or more variables.

Fault model. Nodes may fail-stop and crash, but we assume that the network does not get partitioned. New nodes can join the network. These faults can occur in any finite number, at any time and in any order. A program is *self-healing* if and only if after faults stop occurring the program eventually recovers to a state from where its specification is satisfied.

Problem statement. Design a distributed, local, scalable and self-healing program that constructs a clustering of a network such that:

- a unique node is designated as a clusterhead of each cluster,
- every node in the inner-band of a clusterhead j belongs to j's cluster,
- no node outside the outer-band of a clusterhead j belongs to j's cluster,
- every node belongs to a cluster, and
- no node belongs to multiple clusters.

3.3 FLOC Program

3.3.1 Justification for Stretch-Factor ≥ 2

As an illustration of local self-healing of FLOC, consider Figure 3.4. When j joins the network it is subsumed by one of its neighboring clusters as j is within 2 units of the clusterhead L, thus leading to local healing.

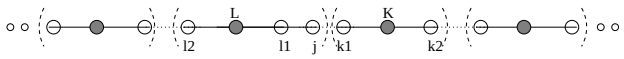


Figure 3.4: New node j joins one of its neighboring clusters.

Furthermore, Figure 3.5 illustrates how FLOC locally self-heals when all clusters are of radius 2 and a new node j joins the network. j elects itself as the clusterhead since it is not within 2 units of the clusterheads of its neighbors $l1$ and $k1$. Nodes $l1$ and $k1$ then join the cluster of j because they are not within 1 unit of their respective clusterheads but are within 1 unit of j. Thus j forms a legitimate cluster as in Figure 3.6.

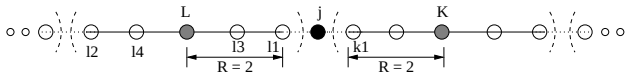


Figure 3.5: $j's$ neighbors are l_1 and k_1.

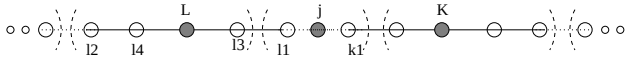


Figure 3.6: j becomes the clusterhead.

3.3.2 Program

Each node j maintains only two variables, *status* and *cluster_id*, for the FLOC program. *j.status* has a domain of {*idle, cand, c_head, i_band, o_band*}. As a shorthand, we use $j.x$ to denote $j.status = x$. *j.idle* is true when j is not part of any cluster. *j.cand* means j wants to be a clusterhead, and *j.c_head* means j is a clusterhead. *j.i_band* (respectively *j.o_band*) means j is an inner-band (resp. outer-band) member of a clusterhead; *j.cluster_id* denotes the cluster j belongs to. Initially for all j, $j.status = idle$ and $j.cluster\_id = \perp$.

FLOC program consists of 6 actions as seen in Figure 3.8.

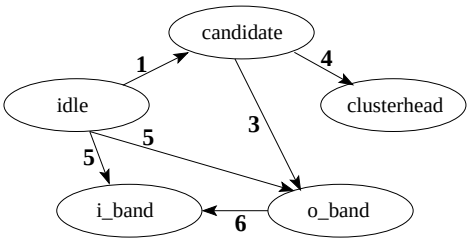


Figure 3.7: The effect of actions on the *status* variable.

Action 1 is enabled when a node j has been *idle* for some random wait-time chosen from the domain $[0 \ldots T]$. Upon execution of action 1, j becomes a *candidate* for becoming a clusterhead, and broadcasts its candidacy.

Action 2 is enabled at an i-band node of an existing cluster when this node receives a candidacy message. If this recipient node determines that it is also in the i-band of the new candidate, it replies with a conflict message to the candidate and attaches its cluster-id to the message. We use a random wait-time from the domain $[0 \ldots t]$ to prevent several nodes replying at the same time so as to avoid collisions.

Action 3 is enabled at j when j receives a conflict message in reply to its candidacy announcement. The conflict message indicates that if j forms a cluster its i-band will overlap with the i-band of the sender's cluster. Thus, j gives up its candidacy and joins the cluster of the sender node as an o-band member.

Action 4 is enabled at j if j does not receive a conflict message to its candidacy within a pre-defined period Δ. In this case j becomes a clusterhead, broadcasts this decision with $c\_head\_msg_j$.

Action 5 is enabled at all the idle nodes that receive a c_head_msg. These nodes determine whether they are in the i-band or o-band of the sender, adjust their status accordingly, and adopt the sender as their clusterhead.

Action 6 is enabled at an o_band node j when j receives a c_head_msg from a clusterhead i of another cluster. If j determines that j falls in the i-band of i, j joins i's cluster as an i_band member.

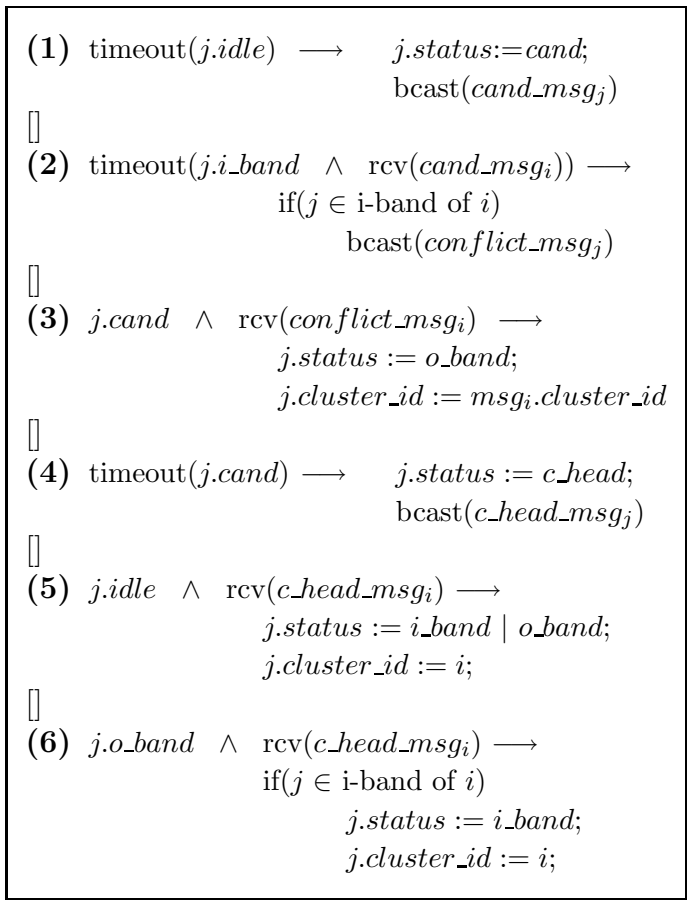


Figure 3.8: Program actions for j.

3.3.3 Analysis

The candidacy period for a node can last at most Δ time, and we require that the election of a clusterhead is completed in an atomic manner: If two nodes that are less than 2 units apart become candidates concurrently, both may succeed and as a result the i-bands of the resultant clusters could be overlapping. To avoid this case with a high probability, the domain T of the timeout period for action 1 should be large enough to ensure that no two nodes that are less than 2 units apart have idle-timers that expire within Δ time of each other.

Note that T depends only on the local density of nodes and is independent of the network size. Hence, it is sufficient to experiment with a representative small portion of a network to come up with a T that avoids collusions of clusterhead elections with a high probability. For the rare cases where the atomicity requirement for elections is violated, our additional actions presented in Section 3.5 reassert the solid-disc clustering property.

Theorem 3.1. Regardless of network size, FLOC produces a clustering of nodes within constant time $T + \Delta$.

Proof. An action is enabled at every node within at most T time: if no other action is enabled in the meanwhile, action 1 is enabled within T time.

From Figure 3.7 it is easy to observe that once an action is enabled at a node j, j is assigned to a cluster within at most Δ time: If the enabled action is 5, then j is assigned to a cluster instantaneously. If the enabled action is 1, then one of actions 3 or 4 is enabled within at most Δ time, upon which j is assigned to a cluster immediately.

Also note that once j is assigned to a cluster (i.e. $j.status \in \{c\_head, i\_band, o\_band\}$) no further action can violate this property. Only actions 2 and 6 can be enabled at j: Action 2 does not change $j.status$, and action 6 changes $j.status$ from *o_band* to *i_band*, but j is still a member of a cluster (in this case a closer cluster).

Thus, every node belongs to a cluster within $T + \Delta$. Since *cluster_id* contains a single value at all times, and no node belongs to multiple clusters.

Furthermore, when the atomicity of elections is satisfied, actions 2, 3, and 6 ensure that the clustering satisfies the solid-disc property: If there is a conflict with the i-band of a candidate j and that of a nearby cluster, then j is notified via action 2, upon which j becomes an *o_band* member of this nearby cluster via action 3. If there is no conflict, j becomes a clusterhead and achieves a solid-disc by dominating all the nodes in its i-band. The *o_band* members of other clusters that fall in the i-band of j join j's cluster due to action 6. □

Theorem 3.2. The number of clusters constructed by FLOC is within 3-folds of the minimum possible number.

Proof. A partitioning of the network with minimum number of clusters is achieved by tiling hexagonal clusters of radius 2 (and circular radius $\sqrt{3}$). The worst case construction, where FLOC partitions the network with maximum number of clusters, is achieved by tiling hexagonal clusters of radius $2/\sqrt{3}$ (and circular radius 1). In this worst case, the number of clusters constructed by FLOC is 3 times the minimum possible number. □

3.3.4 Discussion

After clustering, a node can be in the i-band of at most one clusterhead. A clusterhead has all the nodes in its i-band as its members and some from its o-band. During a convergecast (data aggregation) to the clusterhead, the messages from o-band members may or may not reach the clusterhead directly. If a message from an o-band member is tagged as important, it may be relayed by an i-band member upon detection of a missing acknowledgement from the clusterhead—the i-band members can hear both the clusterhead and the o-band members reliably. Also, the o-band members do not need to hear the clusterhead every time, the i-band members may suffice for most operations. If the clusterhead is sending an important message that needs to reach all members, in order for the o-band members to also receive it reliably, the i-band members may relay this message when they detect missing acknowledgements from nearby o-band members.

Optimization. Ideally, we want that a conflict is first reported by a node that is closest to the candidate, so that the candidate, upon aborting its candidacy, can join this closest cluster. Another advantage of selecting the notifier to be closest to the candidate is that, then the conflict message of the notifier is overheard by as many nodes within the i-band of the candidate, upon which these overhearing nodes can decide that there is no need to report a conflict again. This way communication- and, hence, energy-efficiency is achieved.

One way to choose the closest notifier is to set t at a notifier node to be inversely proportional to the distance from the candidate. If an underlying localization service is not available, the same effect can be achieved by setting t inversely proportional

with respect to the received signal strength of the candidacy message. A notifier sets t smaller the higher the received signal strength of the candidacy message at that notifier.

3.4 Self-Healing

In this section, we discuss the local self-healing properties of our clustering service.

Node failures. FLOC is inherently robust to failures of cluster members (non-clusterhead nodes), since such failures do not violate the clustering specification in Section 3.2.

However, failure of a clusterhead leaves its cluster members orphaned. In order to enable the members to detect the failure of the clusterhead, we employ heartbeats. The clusterhead periodically broadcasts a *c_head_msg*. If the lease at a node j expires, i.e., j fails to receive a heartbeat from its clusterhead within the duration of a lease period, L, then j dissociates itself from the cluster by setting $j.status := idle$ and $j.cluster\_id := \perp$. While setting the idle-timer, j adds L to the selected random wait time so as not to become a candidate before all the members can detect the failure of the clusterhead.

After a clusterhead failure, all the cluster members become *idle* within at most L time. After this point, the dissolved members either join neighboring clusters as o-band members, or an eligible candidate unites these nodes in a new cluster within $T + \Delta$ time. Due to our selection of $m \geq 2$, this is achieved in a local manner.

The lease for o-band nodes should be kept high. Since they receive only a percentage of the heartbeats they may make mistakes for small values of L. Keeping the lease period high for the o-band nodes does not affect the performance significantly,

because the o-band nodes are moldable: Even if they have misinformation about the existence of a clusterhead, the o-band nodes do not hinder new cluster formation, and even join these clusters if they fall within the i-band of these clusterheads.

L is tunable to achieve faster stabilization or better energy-savings.

Node additions. FLOC requires that nodes wait for some random time (chosen from $[0 \ldots T]$) before they can become a candidate. Some of the newly added nodes receive a heartbeat (c_head_msg) from a nearby clusterhead within their initial waiting period and join the corresponding cluster as an i_band or o_band member. Those nodes that fail to receive a heartbeat message within their determined waiting times become candidates, and either form their own clusters (via action 2), or receive a conflict message from an i_band member of a nearby cluster and join that cluster (via action 3).

3.5 Extensions to the Basic FLOC Program

Choosing a sufficiently large T guarantees the atomicity of elections and, hence, the solid-disc clustering. Here we present some additional actions to ensure that the solid-disc property is satisfied even in the statistically rare cases where atomicity of elections are violated.

Consider a candidate i and an idle node k that is within 2 units of i. If k's $idle$ timer expires before i's election is completed (i.e., within Δ time of i's candidacy announcement), then atomicity of elections is violated. Even though there exists a node j that is within the i-bands of both i and k, both candidates may succeed in becoming clusterheads: Since k's candidacy announcement occurs before i's c_head_msg, action 2 is not enabled at j and j does not send a $conflict\_msg$ to k.

Our solution is based on the following observation. Since i broadcasts its *cand_msg* earlier than that of k and since a broadcast is an atomic operation in wireless sensor networks: i's broadcast is received at the same instant by all the nodes within i's i-band. These i-band nodes can be employed for detecting a conflict if a nearby node announces candidacy within Δ of i's candidacy.

To implement our solution we introduce a boolean variable *lock* to capture the states where an idle node j is aware of a candidacy of a node that is within unit distance to itself. The value of $j.lock$ is material only when $j.status = idle$. Our solution consists of 4 actions.

Action 7 is enabled when an idle node j receives a candidacy message. If j determines that j is in the i-band of the candidate, j sets *lock* as *true*.

Action 8 is enabled when an idle and locked node j receives a candidacy message. If j determines that it is also in the i-band of this new candidate, it replies with a "potential conflict" message to the candidate.

Action 9 is enabled when a node receives a "potential conflict" message as a reply to its candidacy announcement. In this case the node gives up its candidacy and becomes idle again. This time, to avoid a lengthy waiting, the node selects the random wait-time from the domain $[0...T/2]$.

Action 10 is enabled if an idle j remains locked for Δ time. Expiration of the Δ timer indicates that the candidate that locked j failed to become a leader: since otherwise j would have received a *c_head_msg* and $j.status$ would have been set to *i_band*. So as not to block future candidates j removes the lock by setting $j.lock := false$.

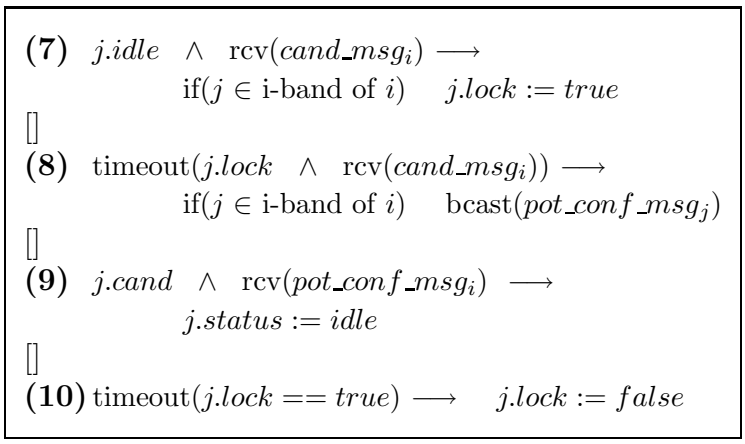


Figure 3.9: Additional actions for j.

Note that these additional actions are applicable only in the statistically rare violations of atomicity of elections; they do not cure the problem for every case. If T is chosen too small, there may be some pathological cases where there is a chain of candidates whose i-bands overlap with each other that results in the deferring of all candidates in the chain. These chains should be avoided by choosing a large enough T.

3.6 Simulation and Implementation Results

In this section we analyze, through simulations and experiments, the tradeoffs between smaller T and the quality of clustering, and determine a suitable value for T that a fast completion time without compromising the quality of the resulting clustering. We also analyze the scalability of FLOC with respect to network size.

3.6.1 Simulation

For our simulations, we use Prowler [97], a MATLAB based, event-driven simulator for wireless sensor networks. Prowler simulates the radio transmission, propagation, and reception delays of Mica2 motes [54], including collisions in ad-hoc radio networks, and the operation of the MAC-layer.

Our implementation of FLOC under Prowler is per node and is a message-passing distributed program. Our code is available from `www.cis.ohio-state.edu/~demirbas/floc/`. In our simulations, we use a grid topology for simplicity (note that FLOC is applicable for any kind of topology and does not require a uniform distribution of nodes). In the grid, each node is unit distance away from its immediate North, South, East, and West neighbors. We use a signal strength of 1 and $m = 2$; the i-band neighbors are the nodes with Received Signal Strength Indicator (RSSI) > 0.5, and the o-band neighbors have $RSSI > 0.2$. It follows that immediate N, S, E, W neighbors are i-band neighbors, and immediate diagonal neighbors and 2-unit distance N, S, E, W neighbors are o-bound neighbors. Thus the degree of a node in our network is between 4 and 12.

Below we analyze the tradeoffs involved in the selection of T; for this part we use a 10-by-10 grid (as described above) for the simulations. Then, we consider larger networks (up to 25-by-25 grids) and investigate the scalability of the performance of FLOC with respect to network size. We repeat each simulation 10 times and take the average value from these runs. In all our graphs, the error bars denote the standard deviation in our data. Due to MAC layer delays, the average transmission time for a packet is around 25 msec. Thus, we fix $t = 50$ msec and $\Delta = 200$ msec for our simulations.

Tradeoffs in the selection of T. Using a small value for T allows a shorter completion time for FLOC as shown in Figure 3.10. However, a small value for T also increases the probability of violation of atomicity of elections; Figure 3.11 shows that while T decreases the number of violations of atomicity of elections increases.

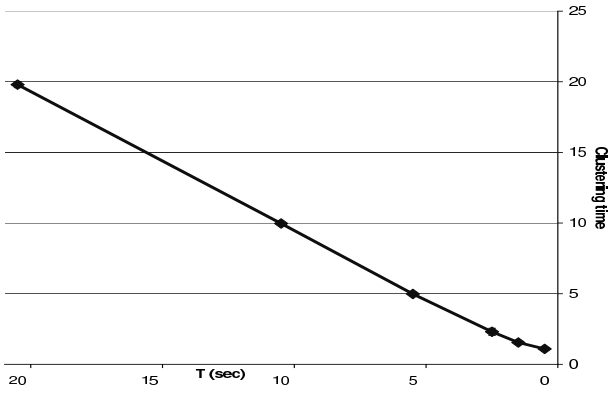


Figure 3.10: Completion time versus T

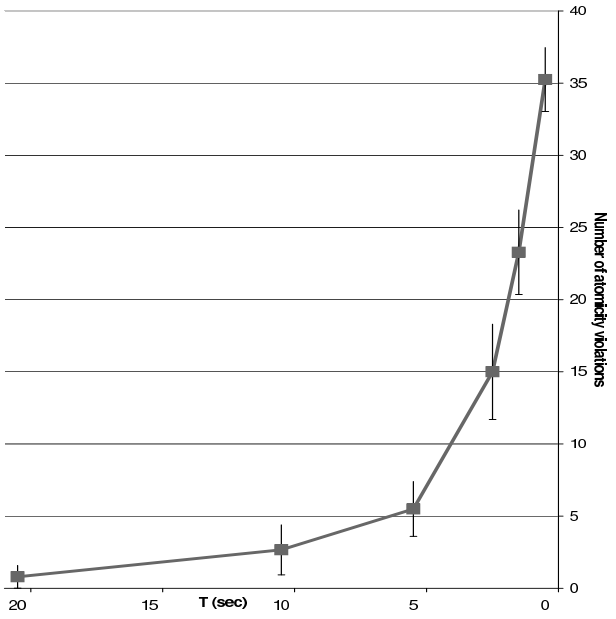


Figure 3.11: Number of atomicity violations versus T

Ideally, we want the elections to be completed in an atomic manner. For up to some number of atomicity violations, our extra actions in Section 3.5 enable successful solid-disc clustering. However, for small values of T ($T < 5$ sec) several nodes declare their candidacy around the same times, and we encounter a sharp increase in the number of messages sent and the number of nodes sending messages as shown in Figure 3.12. This leads to network traffic congestion and loss of messages due to collisions. For $T = 2$ the number of reception of collided messages are 20% of the total messages received. This collision rate climbs to 30% for $T = 1$, and 55% for $T = 0.5$. Due to these lost messages, for $T < 5$, we observe deformities in the shape of the clusters formed; the solid-disc clustering property is violated. For example, for $T = 0.5$ half of the clusters formed are single node clusters. As a result, we observe an increase in the number of clusters formed as shown in Figure 3.13.

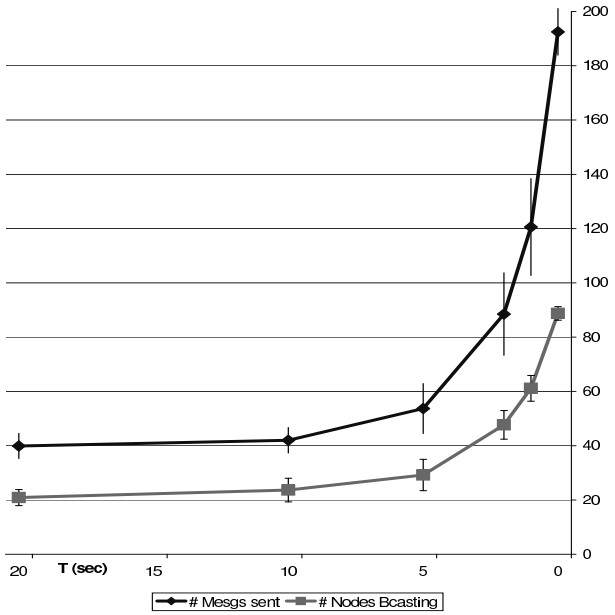


Figure 3.12: Messages sent versus T

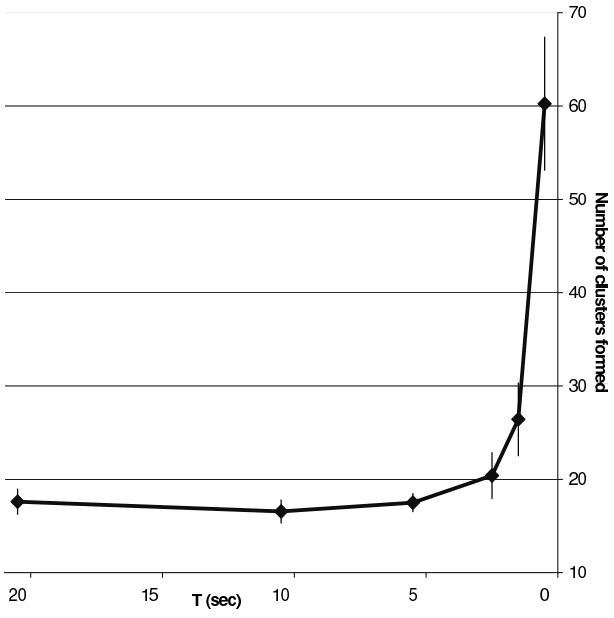


Figure 3.13: Number of clusters formed versus T

To achieve a quick completion time while not compromising the quality of the resulting clustering, we choose $T = 5$ sec in our FLOC program –and for the rest of this section. We observe that for $T = 5$ the solid-disc clustering property is satisfied by every run of the FLOC program. Figure 3.14 shows a resulting partitioning on a 10-by-10 grid. The arrow at a node points to its respective clusterhead. There are 16 clusters; each clusterhead contains at least its i-band neighbors as it members, that is, solid-disc clustering is observed.

Scalability with respect to network size. In Theorem 1, we showed that the completion time of FLOC is unaffected by the network size. To corroborate this result empirically, we simulated FLOC with $T = 5$ for increasing network size of up to 25-by-25 nodes while preserving the node density. Figure 3.15 shows that the clustering is achieved in 5 sec regardless of network size.

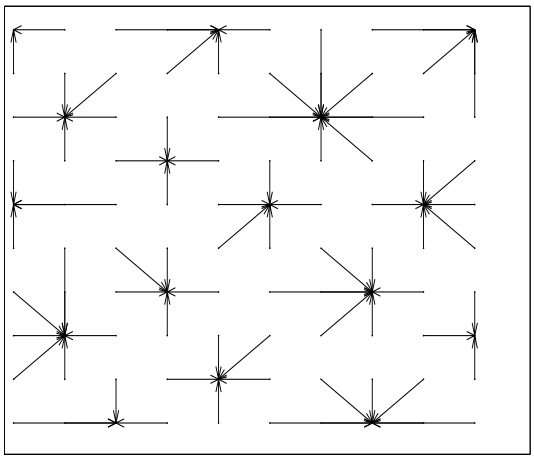


Figure 3.14: Clusters formed by FLOC on a 10-by-10 grid.

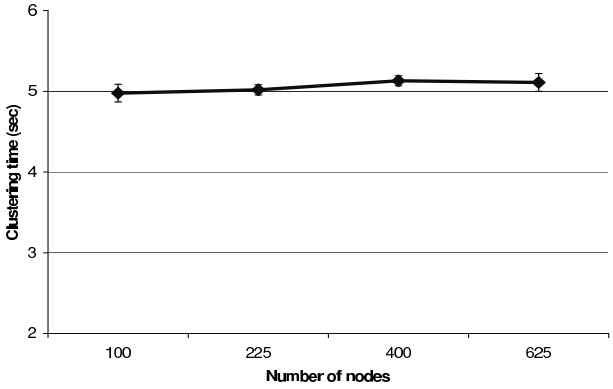


Figure 3.15: Completion time versus network size

We also investigated the average number of clusters constructed (NCC) by FLOC with respect to increasing network size. An interesting observation is that, NCC for a given N is predictable; the variance is very small as seen in Figure 3.16. Since clusters have, on average, around 6 members, $N/6$ gives NCC for our grid topology networks.

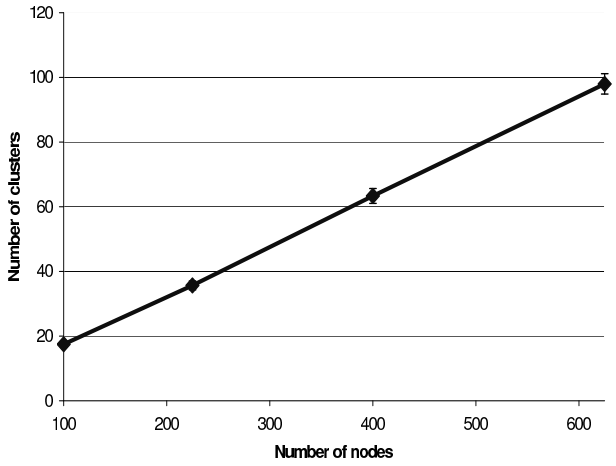


Figure 3.16: Number of clusters formed versus network size

For a grid of 25-by-25, FLOC constructs around 100 clusters. In the theoretical best case, an omniscient centralized partitioning scheme (see Theorem 2) could tile this grid with 60 hexagons (with circular radius of $\sqrt{3}$ and hexagonal radius of 2). That is, in practice FLOC has an overhead of only 1.67 when compared with the best scheme. Note that, in Theorem 2, we have determined that NCC for FLOC is always within 3-folds of this best scheme.

3.6.2 Implementation

We implemented FLOC on the Mica2 [102] mote platform using the TinyOS [55] programming suite. Our implementation is about 500 lines of code and available from `www.cis.ohio-state.edu/~demirbas/floc/`.

The Mica2 motes use Chipcon [24] radio CC1000 for transmission. RSSI at a mote can be obtained using the CC1000 radio interface in the TinyOS radio stack: RSSI varies from -53dB to -94dB, the radio interface encodes this into a 16 bit integer value —the lower the value the higher the signal strength. By experimenting at an

outdoor environment and comparing power level and reliable range of reception we chose a transmission power of 7, from a range of 1 to 255. At a power level 7, we obtain reliable reception up to 15 feet with RSSI ranging from 0 to 140. By selecting appropriate thresholds for RSSI, we took $m=2$ and divided this 15 feet distance into two equal halves as i-band range and o-band range: we considered RSSI between 0-80 as i-band and 80-140 as o-band.

Figure 3.17: 5-by-5 grid topology deployment

We performed our experiments at an outdoor parking lot; Figure 3.17 shows a picture of our deployment. To mimic our simulation topology settings, we arranged 25 Mica2 motes in a 5-by-5 grid where each mote is 6 feet away from its immediate North, South, East, and West neighbors. From our signal strength settings it follows that, ideally, immediate N, S, E, W neighbors are i-band neighbors, and immediate diagonal neighbors and 2-unit distance N, S, E, W neighbors are o-bound neighbors. Based on our simulation results, to achieve a quick completion time while avoiding network contention, we chose $T=5$ sec, $\Delta=200$ msec.

In our set up, we placed a laptop in the center of the network to collect status reports from the motes: After the clustering is completed, every mote temporarily sets its transmission power to maximum level and broadcasts a status report. This report indicates the completion time of the clustering program at the respective mote, and whether the mote is a clusterhead, i-band, or o-band member of a cluster. In order to avoid collisions, these reports are spread in time.

We performed over 20 experiments with these settings. We observed the average number of clusters formed to be 4. The cluster sizes were reasonably uniform, the average number of motes per cluster was 6. The average completion time was 4.5 seconds.

When we increased the inter node spacing to 8 feet, with the same settings for signal strength measurements, the number of clusters increased to an average of 6 as expected. The average completion time was again 4.5 seconds.

We observed in our experiments that, due to the nondeterministic nature of wireless radio communication, the i-band/o-band membership determination using RSSI is not always robust. Transmitting candidacy and clusterhead messages 3 times, and using the average RSSI from the corresponding 3 receptions would make the i-band/o-band determination more robust. Alternatively, as we discussed in Section 3.2, a connectivity service or localization service can be employed for i-band/o-band membership determination.

3.7 Related Work

Several protocols have been proposed recently for clustering in wireless networks [4, 19, 21, 50, 84].

Max-Min D-cluster algorithm [4] partitions the network into d-hop clusters. It does not guarantee solid-disc clustering and in the worst case, the number of clusters generated may be equal to the number of nodes in the network (for a connected network).

Clubs [84] forms 1-hop clusters: If two clusterheads are within 1-hop range of each other, then both the clusters are collapsed and the process of electing clusterheads via random timeouts is repeated. Clubs does not satisfy our unit distance solid-disc clustering property: clusterheads can share their 1-hop members. Also, in contrast to Clubs, FLOC does not collapse any cluster once it is formed. FLOC resolves contentions by delaying the latter candidates from becoming clusterheads.

LEACH [50] also forms 1-hop clusters. The energy load of being a clusterhead is evenly distributed among the nodes by incorporating randomized rotation of the high-energy clusterhead position among the nodes. Nodes elect themselves as clusterheads based on this probabilistic rotation function and broadcast their decisions. Each non-clusterhead node determines its cluster by choosing the clusterhead that requires the minimum communication energy. LEACH does not satisfy our solid-disc property: Not all nodes within 1-hop of a clusterhead j belongs to j. Hence, in LEACH the clusterheads are susceptible to network contention induced by members of other clusters. The authors [50] suggest using different Code Division Multiple Access (CDMA) spreading codes for each cluster to solve this problem, however, for most sensor network platforms (including Mica2) CDMA mechanism is not available. FLOC complements LEACH since it addresses the network contention problem at the clusterheads by constructing solid-disc clusters. Moreover, LEACH style load-balancing is readily applicable in FLOC by using the above mentioned probabilistic

rotation function for determining the waiting-times for the candidacy announcements at the nodes. By adopting FLOC, it is also possible to guarantee a tight upperbound on the number of clusters formed.

The algorithm in [19] first finds a rooted spanning tree of the network and then forms desired clusters from the subtrees. It gives a bound on the number of clusters constructed and the convergence time is of the order of the diameter of the network. It is locally fault-tolerant to node failures/joins but may lead to re-clustering of the entire network for some pathological scenarios.

For a given value of R, the algorithm in [21] constructs clusters such that all the nodes within $R/2$ hops of a clusterhead belong to that clusterhead and the farthest distance of any node from its clusterhead is 3.5R hops. With high probability, a network cover is constructed in $O(R)$ rounds; the communication cost is $O(R^3)$.

In an earlier technical report [83], we have presented –under a shared memory model– a self-stabilizing clustering protocol, LOCI, that partitions a network into clusters of bounded physical radius $[R, mR]$ for $m \geq 2$. LOCI achieves a solid-disc clustering with radius R. Clustering is completed iteratively within $O(R^4)$ rounds.

3.8 Chapter Summary

The properties of FLOC that make it suitable for large scale wireless sensor networks are its: *(1)* locality, in that each node is affected only by nodes within m units, *(2)* scalability, in that clustering is achieved in constant time independent of network size, and finally *(3)* self-healing capability, in that it tolerates node failures and joins locally within m units.

Through simulations and experiments with actual deployments, we analyzed the tradeoffs between completion time and the quality of the resulting clustering, and

suggested suitable values for the domain, T, of the randomized candidacy timer to achieve a fast completion time without compromising the quality of the clustering. Since in FLOC each node is affected only by nodes within m units, it is sufficient to experiment with a representative small portion of a network to determine suitable values for T.

As part of future work, we are planning on integrating FLOC in our "Line in the Sand" (LITeS) tracking service [7] to achieve scalable and fault-local clustering. As part of the DARPA/Network Embedded Systems Technology project, our research group has already deployed LITeS over a 100-node sensor network across a large terrain and achieved detection, classification, and tracking of various types of intruders (e.g., persons, cars) as they moved through the network. We are also investigating the role of geometric, local clustering in designing efficient data structures for evaluation of spatial queries in the context of sensor networks.

CHAPTER 4

RELATED WORK ON FAULT-CONTAINMENT AND FAULT-LOCAL HEALING

In this chapter we provide a brief survey of work on fault-containment in Section 4.1 and work on fault-local stabilization in Section 4.2.

4.1 Fault-containment

Fault-containment is a highly desirable property for fault-tolerance systems, and it has received a lot of attention in the fault-tolerance literature. Here we overview previous work on fault-containment. We discuss masking fault-tolerance approaches for fault-containment in Section 4.1.1. In Section 4.1.2 we mention approaches that contain faults within software module boundaries, and in Section 4.1.3 approaches that bound faults within predefined sections of a network.

4.1.1 Fault-containment through masking

One way of achieving fault-containment is through design of masking fault-tolerance: A system is masking tolerant if in the presence of faults, it always satisfies its safety properties and, when faults stop occuring, it eventually resumes satisfying its liveness properties. Since all faults are masked immediately, a masking fault-tolerant system is trivially fault-containing.

Masking fault-tolerance is a strong property and hence it is applicable for a limited class of faults, such as single node failure, message loss, etc. Also, since masking tolerance is overly ambitious, the cost of masking tolerance may be too high a price to pay in wireless sensor networks, where energy is a precious commodity, and in real-time applications, where the freshness of data is as important as the accuracy of the data. These concerns led researchers to investigate other approaches for fault-containment. In this chapter, we focus on these approaches, and relegate an overview of work on masking fault-tolerance to Section where we survey previous work on fault-tolerance design methods.

4.1.2 Fault-containment within prespecified modules of the system

Below, we mention two popular schemes, *Sandboxing* and *Input/Output checking*, for achieving fault-containment within prespecified modules of the system.

Sandboxing. The idea behind sandboxing approach is simple: the user program is isolated in a sandbox where it can execute without harming anything outside the sandbox. The boundaries of this notional box limit the scope of a malicious program to cause damage to the computer system as a whole. In the Java system [58], for example, most applets are run in a software sandbox. A virtual machine [106], a software abstraction of a machine on top of another machine, is also another example of the sandboxing approach.

Sandbox consists of user code and data segment. A typical sandbox implementation employs the following techniques:

- configuring the memory manager unit to throw an exception for accesses outside of sandbox,

- rewriting the binary to mask off higher order bits on addresses to keep them within the sandbox, and
- redirecting system calls through a protected jump table to an arbitrator.

Input/output checking. Input/output checking allows a system to limit the impact of manifested faults to some predefined system module boundaries. In [94], input/output checking techniques for fault containment are formatted and presented as design patterns. The presented fault containment patterns are: the Input Guard pattern which confines an error outside the guarded system boundaries; the Output Guard which confines an error inside the guarded system boundaries; and the Fault Container pattern which is the fault tolerant counterpart of the well-known Adapter pattern and which combines the properties of the Input Guard and Guard patterns.

4.1.3 Fault-containment within prespecified system boundaries

In [42], the authors show that by adding structure and sacrificing full distribution it is possible to improve the fault-containment of self-stabilizing algorithms. To demonstrate this, they investigate an application of the general principle of "introducing structure" to the area of self-stabilizing spanning-tree construction. By doing this, they show that it is possible to transform an arbitrary self-stabilizing spanning-tree algorithm into one with increased efficiency and fault-containment properties. After adding the fixed structure, faults can only lead to perturbations within the algorithm instances in which they happen. This is in contrast to the case where a standard spanning-tree algorithm runs in the entire network: a single fault (e.g., of the root) can lead to a global reconfiguration. However, the level of fault-containment

again depends on the distribution of algorithm instances to processing elements: if one processing element participates in all algorithm instances then a failure of this node may also cause global disruption.

A hierarchical error detection and containment framework for a Software Implemented Fault Tolerance (SIFT) layer of a distributed system is proposed in [18]. The design and implementation of a software-based distributed signature monitoring scheme is central to the proposed four-level hierarchy: process level, node level, group level, and across groups. The paper reports a substantial increase in availability due to the detection framework and help in understanding the trade-offs between overhead and coverage for different combinations of techniques.

4.2 Fault-local stabilization

The area of fault-containment of self-stabilizing algorithms has received growing interest since it was first introduced in [45]. Below we present an overview of some previous work on fault-local stabilization.

4.2.1 Solutions that are local but not fault-local

A notion of local correction was suggested in [14] in the context of self stabilization. The meaning of locality there is that each node can act locally to correct a state of an algorithm. However, if the corrected algorithm is global, then the function computed by the corrected algorithm it can be output only after O(n) (number of nodes) or O(Diarneter) (diameter of the network). (In fact the example used in [14] for the corrected algorithm is the global reset algorithm.)

Another concept worth mentioning is snap stabilization [30]: A system is called snap-stabilizing if its behavior stabilizes to its specification in 0 time. Clearly, snap

stabilization is possible only for a certain class of task specifications, that allow a faulty node to be considered externally correct even at the time of the fault (broadcast does not satisfy this requirement). On the other hand, no known snap-stabilizing algorithm is error confined.

4.2.2 Solutions that are fault-local in time but not in work

Kutten and Peleg [68] introduce the notion of fault mending, which addresses the issue of relating repair time to fault severity; their method is based on state replication and is self-stabilizing only in a synchronous system.

In [1], a local stabilizer protocol that converts a distributed algorithm into a synchronous self-stabilizing algorithm with local monitoring and repairing properties is presented. Whenever the self-stabilizing version enters an inconsistent state, the inconsistency is detected, in O(1) time, and the system state is repaired within time proportional to the largest diameter of a faulty region. This method is also based on an underlying state replication protocol, and hence, even though the recovery-time is fault-local, the work done is not local.

In [17], a broadcast protocol is proposed to contain observable variables in the presence of state corruptions, but the protocol allows for global propagation of internal protocol variables.

4.2.3 Solutions that are fault-local but that assume a restricted fault-model

The notion of fault containment within the context of stabilization is formalized first in [45]; algorithms were proposed to contain state-corruption of a single node in a stabilizing spanning tree protocol.

Dolev and Herman [37] construct self-stabilizing protocols that guarantee safe convergence from states that arise from legitimate states perturbed by limited topology changes.

Some other examples of self-stabilizing algorithms that guarantee rapid convergence from states that arise from legitimate states due to single-process faults are presented in the context of leader election problem [44] and mutual exclusion problem [53].

4.2.4 Fault-local stabilizing solutions

In [86] fault-containment of Byzantine nodes have been studied for dining philosophers and graph coloring algorithms. In [89] a dining-philosophers algorithm with crash-locality 1 is presented under a partially synchronous model. These work require the intrinsic locality of the problem to be constant and is too limiting for problems such as tracking and routing whose locality are not constant.

A protocol that achieves fault-local stabilization in shortest path routing is presented in [12]. To achieve fault-containment the protocol uses containment actions that are a constant time faster than the fault-intolerant program actions. In contrast to [12], we do not have a privileged set of containment actions in STALK; the program actions serve to this end. We enable fault-containment in a hierarchy-based manner by suitably varying the speed of actions (through the use of process timers) as per the level of the hierarchy they are executed at.

PART II
SPECIFICATION-BASED DESIGN OF SELF-HEALING

CHAPTER 5

SPECIFICATION-BASED DESIGN METHOD

5.1 Introduction

Research in self-healing, more specifically in self-stabilization [36, 39, 46, 52], has traditionally relied on the availability of a complete system implementation. The standard approach uses knowledge of all implementation variables and actions to exhibit an "invariant" condition such that if the system is properly initialized then the invariant is always satisfied and if the system is placed in an arbitrary state then continued execution of the system eventually reaches a state from where the invariant is always satisfied. The apparently intimate connection between stabilization and the details of implementation has raised the following serious concerns: (1) Stabilization is not feasible for many applications whose implementation details are not available, for instance, closed-source applications. (2) Even if implementation details are available, stabilization is not scalable as the complexity of calculating the invariant of large implementations may be exorbitant. (3) Stabilization lacks reusability since it is specific to a particular implementation.

Towards addressing these concerns, in this chapter, we show that system stabilization may be achieved without knowledge of implementation details. We eschew

"whitebox" knowledge—of system implementation—in favor of "graybox" knowledge—of system specification—for the design of stabilization. Since specifications are typically more succinct than implementations, specification-based design of fault-tolerance offers the promise of scalability when the design effort for adding fault-tolerance is proportional to the size of the system.. Also, since specifications admit multiple implementations and since system components are often reused, specification-based design of fault-tolerance offers the promise of reusability. Finally, for closed-source situations where exploiting a specification is warranted, specification-based approach allows the design of efficient fault-tolerance in contrast to a blackbox design.

Given a high-level system specification A, the specification-based approach is to design a tolerance wrapper W such that adding W to A yields a fault-tolerant system. The goal is to ensure that for any low-level refinement (implementation) C of A adding a low-level refinement W' of W would also yield a fault-tolerant system.

Note that since the refinements from A to C and W to W' can be done independently, specification-based design enables a posteriori or dynamic addition of fault-tolerance. That is, given a concrete implementation C, it is possible to add fault-tolerance to C as follows:

- First, design an abstract (high-level) tolerance wrapper W using solely an abstract specification A of C, and then
- add a concrete (low-level) refinement W' of W to C.

The goal of specification-based fault-tolerance is not readily achieved for all refinements. The refinements we need for achieving specification-based fault-tolerance should not only preserve fault-tolerance but also have nice composability features so that the refinements from A to C and W to W' can be done independently. In this

chapter we present special classes of refinement, "everywhere refinements", "local-everywhere refinements", and "convergence refinements", that enable specification-based design of stabilization. These refinements ensure that if A composed with W is fault-tolerant, then for any everywhere or convergence refinement C of A adding an everywhere or convergence refinement W' of W would also yield a fault-tolerant system.

Outline of the rest of the chapter. In Section 5.2, we give preliminaries. In Section 5.3 we show that everywhere and convergence refinements are stabilization preserving and in Section 5.4 that they are amenable for specification-based design of stabilization. We make concluding remarks in Section 5.5.

We refer the reader to the full version of our works on everywhere refinement [6] and convergence refinement [32] for several illustrations of our specification-based design method. In those work, we present specification-based design of stabilization for the Ricart-Agrawala and Lamport mutual exclusion algorithms, and a derivation of Dijkstra's stabilizing token-ring algorithms as a refinement of a simple, abstract token-ring algorithm.

5.2 Preliminaries

Let Σ be a state space.

Definition. A *system* S is a finite-state automaton (Σ, T, I) where T, the set of transitions, is a subset of $\{(s_0, s_1) : s_0, s_1 \in \Sigma\}$ and I, the set of initial states, is a subset of Σ.

A computation of S is a maximal sequence of states such that every state is related to the subsequent one with a transition in T, i.e., if a computation is finite there are no transitions in T that start at the final state.

We refer to an abstract system as a *specification*, and to a concrete system as an *implementation*. For now we assume for convenience that the specification and the implementation use the same state space. In Section 5.4.4, we present a generalization that allows the implementation to use a different state space than the specification. Henceforth, let C be an implementation and A a specification.

Definition. C is a *refinement* of A, denoted $[C \subseteq A]_{init}$, iff every computation of C that starts from an initial state is a computation of A.

Definition. C is an *everywhere refinement* [6] of A, denoted $[C \subseteq A]$, iff every computation of C is a computation of A.

Definition. A state sequence c is a *convergence isomorphism* of a state sequence a iff c is a subsequence of a with at most a finite number of omissions and with the same initial and final (if any) state as a.

For instance, $c = s1\ s3\ s6$ is a convergence isomorphism of $a = s1\ s2\ s3\ s4\ s5\ s6$. However, $c = s1\ s3\ s5\ s6$ is *not* a convergence isomorphism of $a = s1\ s2\ s5\ s6$ since c can only drop states in a, and cannot insert states to a. Intuitively, the convergence isomorphism requirement corresponds to the notion of using similar recovery paths: c should use a similar recovery path with a and not any arbitrary recovery path.

Definition. C is a *convergence refinement* of A, denoted $[C \preceq A]$, iff:

- C is a refinement of A,
- every computation of C is a convergence isomorphism of some computation of A.

Note that convergence refinements are more general than everywhere refinements: $[C \subseteq A] \Rightarrow [C \preceq A]$, but not vice versa.

A fault is a perturbation of the system state. Here, we focus on transient faults that may arbitrarily corrupt the process states. The following definition captures a standard tolerance to transient faults.

Definition. *C is stabilizing to A* iff every computation of C has a suffix that is a suffix of some computation of A that starts at an initial state of A.

This definition of stabilization allows the possibility that A is stabilizing to A, that is, A is self-stabilizing.

5.3 Stabilization Preserving Refinements

Refinement tools such as compilers, program transformers, and code optimizers generally do not preserve the fault-tolerance properties of their input programs. Consider, for example, a program that is trivially tolerant to the corruption of a variable `x` in that it eventually ensures `x` is always 0.

```
int x=0;
while(x==x)  {
   x=0;}
```

The bytecode that a Java compiler produces for this input program is not tolerant.

```
 0   iconst_0
 1   istore_1
 2   goto 7
 5   iconst_0
 6   istore_1
 7   iload_1
 8   iload_1
 9   if_icmpeq 5
12   return
```

If the value of `x` (i.e., the value of the local variable at position 1) is corrupted after line 7 is executed and before line 8 is executed (i.e., during the evaluation of "`x==x`") then the execution terminates at line 12, thereby failing to eventually ensure that `x` is always 0.

As another example, consider the specification of a bidding server component. The server accepts bids during a bidding period via a "`bid(integer)`" method and stores only the highest k bids in order to declare them as winners when the bidding period is over. When the "`bid(`v`)`" method is invoked, the server replaces its minimum stored bid with v only if v is greater than the minimum stored bid. The bidding server is tolerant to the corruption of a single stored bid in that it satisfies the specification for $(k-1)$ out of best-k bids.

Consider now a sorted-list implementation of the bidding server. The implementation maintains the highest k bids in sorted order with their minimum being at the head of the list. When the "`bid(`v`)`" method is invoked on the implementation, it checks whether v is greater than the bid value at the head of the list, and if so, the head of the list is deleted and v is properly inserted to maintain the list sort order. This implementation, while correct with respect to the specification in the absence of faults, does not tolerate the corruption of a single stored bid: If the stored bid at the head of the list is corrupted to be equal to `MAX_INTEGER`, then the implementation prevents new bid values from entering the list, and hence fails to satisfy the specification for $(k-1)$ out of best-k bids.

These examples illustrate that even though an abstract system A is fault-tolerant, it is possible that a refinement C of A may not be fault-tolerant since the extra states introduced in C create additional challenges for the fault-tolerance of C. That is,

C refines A and *A is stabilizing to A*

does not imply that *C is stabilizing to A*.

For a more abstract counterexample, consider Figure 5.1. Here $s0$, $s1$, $s2$, $s3, \ldots$ and s* are states in Σ, and $s0$ is the initial state of both A and C. In both A and C, there is only one computation that starts from the initial state, namely "$s0$, $s1$, $s2$, $s3, \ldots$"; hence, $[C \subseteq A]_{init}$. But "s*, $s2$, $s3, \ldots$" is a computation that is in A but not in C. Letting F denote a transient state corruption fault that yields s* upon starting from $s0$, it follows that although A is stabilizing to A if F occurs initially, C is not.

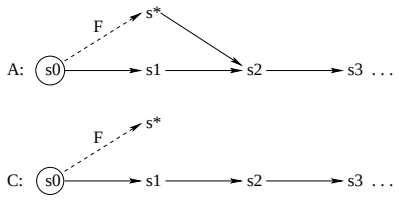


Figure 5.1: $[C \subseteq A]_{init}$

We are therefore motivated to use suitable stabilization preserving refinements in order to enable a specification-based design of stabilization. Next, we present the stabilization preserving properties of everywhere and convergence refinements.

Theorem 5.0 . If $[C \subseteq A]$ and A is stabilizing to B,

then C is stabilizing to B. □

Theorem 5.0 follows immediately from the definitions of stabilization and everywhere refinement.

The requirements for everywhere refinements are sometimes too restrictive. For instance, every computation of the concrete might not be a computation of the abstract since the execution model of the concrete is more restrictive than that of the abstract. One such example is model refinements where a process is allowed to write to the state of its neighbor in the abstract system but not allowed to do so in the concrete system. To address such cases, we consider the more general convergence refinements.

Theorem 5.1 . If $[C \preceq A]$ and A is stabilizing to B,

then C is stabilizing to B. □

Theorem 5.1 follows immediately from the definitions of stabilization and convergence refinement (C can only drop a finite number of states from the computations of A). Theorem 5.1 is the formal statement of the amenability of convergence refinements as stabilization preserving refinements.

5.4 Specification-Based Design of Stabilization

Here we focus on the problem of how to design stabilization to a given implementation C using only its specification A. That is, we want to prove that: If adding a wrapper W to a specification A renders A stabilizing, then adding W to any everywhere or convergence refinement C of A also yields a stabilizing system. We define a wrapper to be a system over Σ and formulate the "addition" of one system to another in terms of the operator $\Box$ (pronounced "box") which denotes the union of automata.

Next, we prove that everywhere and convergence refinements enable specification-based design of stabilization, respectively in Sections 5.4.1 and 5.4.2. In Section

5.4.3 we prove composition theorems about everywhere and convergence refinements. Finally, in Section 5.4.4, we present a generalization of our model that allows the implementation to use a different state space than the specification.

5.4.1 Everywhere refinements

Lemma 5.2 . $([C \subseteq A] \wedge [W' \subseteq W]) \Rightarrow [(C \,[]\, W') \subseteq (A \,[]\, W)]$ □

From the lemma, our goal follows trivially:

Theorem 5.3 (Stabilization via everywhere refinements).
If $[C \subseteq A]$, $A \,[]\, W$ is stabilizing to A, and $[W' \subseteq W]$ then $C \,[]\, W'$ is stabilizing to A. □

Recall that W' and W are designed based only on the knowledge of A and not of C in the specification-based design approach. This results in the reusability of the wrapper for any everywhere implementation of A.

We now focus our attention on distributed systems. The task of verifying everywhere implementation is difficult for distributed implementations, because global state is not available for instantaneous access, all possible interleavings of the steps of multiple processes have to be accounted for, and global invariants are hard to calculate. For effective specification-based design of stabilization of distributed systems, we therefore restrict our consideration to a subclass of everywhere specifications, namely *local everywhere specifications*.

A local everywhere specification A is one that is decomposable into local specifications, one for every process i; i.e., $A = ([]\, i :: A_i)$. Hence, given a distributed implementation $C = ([]\, i :: C_i)$ it suffices to verify that $[C_i \subseteq A_i]$ for each process i. Verifying these "local implementations" is easier than verifying $[C \subseteq A]$ as the

former depends only on the local state of each process and is independent of the environment of each process, thereby avoids the necessity of reasoning about the states of other processes.

Let $A = (\Box\, i :: A_i)$, $C = (\Box\, i :: C_i)$, $W = (\Box\, i :: W_i)$, and $W' = (\Box\, i :: W'_i)$.

Lemma 5.4 . $(\forall i :: [C_i \subseteq A_i]) \Rightarrow [C \subseteq A]$ □

Lemma 5.5 . $((\forall i :: [C_i \subseteq A_i]) \wedge (\forall i :: [W'_i \subseteq W_i])) \Rightarrow [(C \Box W') \subseteq (A \Box W)]$

□

From Lemma 5.5 and Theorem 5.3 , we have

Theorem 5.6 (Stabilization via local everywhere refinements).

If $(\forall i :: [C_i \subseteq A_i])$, $(\forall i :: [W'_i \subseteq W_i])$, and $A \Box W$ is stabilizing to A, then $C \Box W'$ is stabilizing to A. □

Theorem 5.6 is the formal statement of the amenability of local everywhere specifications for specification-based design of stabilization. Again, it is tacit that W'_i and W_i are designed based only on the knowledge of A_i and not of C_i.

5.4.2 Convergence refinements

Lemma 5.7 If $[C \preceq A]$ and $(A \Box W)$ is stabilizing to A

then $[(C \Box W) \preceq (A \Box W)]$.

Proof. This proof consists of two parts. We prove $[(C \Box W) \subseteq (A \Box W)]_{init}$ in the first part, and we prove in the second part that every computation x of $(C \Box W)$ is a convergence isomorphism of a computation x' of $(A \Box W)$.

1. $[C \preceq A] \Rightarrow [C \subseteq A]_{init}$. Thus, every computation of C starting from the initial states is a computation of A, and hence $[(C \Box W) \subseteq (A \Box W)]_{init}$.

2. Any computation x of $(C \ [] \ W)$ can be written as $\cdots - CS_i - WS_i - CS_{i+1} - WS_{i+1} - \cdots$ where CS denotes consecutive states produced by C and WS denotes consecutive states produced by W. Since $[C \preceq A]$, C can only drop states from computations of A. Thus, there exists a computation x' of $(A \ [] \ W)$ of the form $\cdots - AS_i - WS_i - AS_{i+1} - WS_{i+1} - \cdots - A_{init}$ where for all i, CS_i is a convergence isomorphism of AS_i Since $(A \ [] \ W)$ is stabilizing to A, x' has a suffix, A_{init}, that is a suffix of some computation of A that starts from the initial states. Since $[C \preceq A] \Rightarrow [C \subseteq A]_{init}$, x cannot drop any states from x' after $(A \ [] \ W)$ stabilizes to A. That is, x can drop only a finite number of states from x', and hence we conclude that x is a convergence isomorphism of x'. □

Theorem 5.8 If $[C \preceq A]$ and $(A \ [] \ W)$ is stabilizing to A
then $(C \ [] \ W)$ is stabilizing to A.

Proof. The result follows from Lemma 5.7 and Theorem 5.1 . □

Theorem 5.8 states that if a wrapper W satisfies $(A \ [] \ W)$ is stabilizing to A, then, for any C that satisfies $[C \preceq A]$, $(C \ [] \ W)$ is stabilizing to A. In fact, after proving Lemma 5.9 , we prove a more general result in Theorem 5.10 .

Lemma 5.9 If $[W' \preceq W]$ and $(A \ [] \ W)$ is stabilizing to A
then $(A \ [] \ W')$ is stabilizing to A.

Proof. Note that $[W' \preceq W]$ and "$(A \ [] \ W)$ is stabilizing to A" implies $[A \ [] \ W' \preceq A \ [] \ W]$. (This proof is similar to the proof of Lemma 5.7 , and hence, is not included here.) The result follows from the above via Theorem 5.1 . □

Theorem 5.10 If $[C \preceq A]$ and $(A \ [] \ W)$ is stabilizing to A
then $(\forall W' : [W' \preceq W] : (C \ [] \ W')$ is stabilizing to $A)$.

Proof. Given $[W' \preceq W]$ and $(A \Box W)$ is stabilizing to A, we get $(A \Box W')$ is stabilizing to A from Lemma 5.9 . Since $[C \preceq A]$, from Lemma 5.7 we get $[(C \Box W') \preceq (A \Box W')]$. The result follows via Theorem 5.1 . □

Theorem 5.10 is the formal statement of the amenability of convergence refinements for specification-based design of stabilization: If W provides stabilization to A, then any convergence refinement W' of W provides stabilization to every convergence refinement C of A.

5.4.3 Compositionality of everywhere and convergence refinements

Here we prove composition theorems about everywhere and convergence refinements. In contrast to previous work on composition of fault-tolerance, these theorems are applicable for the general case of composition and are not limited to special cases such as layering composition.

Composition theorem for everywhere refinements. Composition theorem for everywhere refinements is simple:

Theorem 5.11 *(Composition of everywhere refinements).*

$$[C \subseteq A] \wedge [D \subseteq B] \wedge [I' \subseteq I]$$
$$\Rightarrow \quad [C \Box D \Box I' \subseteq A \Box B \Box I]$$

□

Everywhere refinements are very easy to compose and thus specification-based approach for composition of fault-tolerance readily applies for everywhere refinements.

Composition theorem for convergence refinements. Convergence refinements are weaker than everywhere refinements and thus they do not compose as cleanly as everywhere refinements.

(Example): $[C \preceq A] \Rightarrow [C \,[]\, B \preceq A \,[]\, B]$ is not a theorem for convergence refinements. Consider the following counterexample.

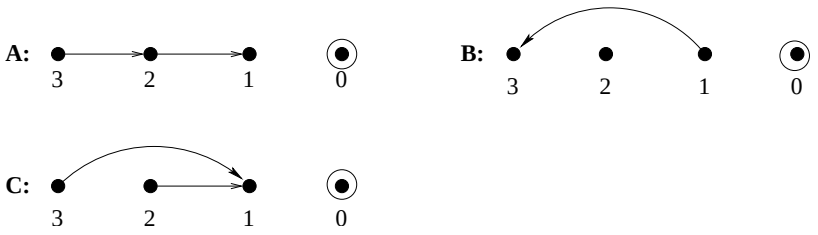


Figure 5.2: $[C \preceq A]$

Let A, C, and B be as in Figure 5.2. State 0 is the initial state for A, C and B, thus $[C \subseteq A]_{init}$ and $[C \preceq A]$.

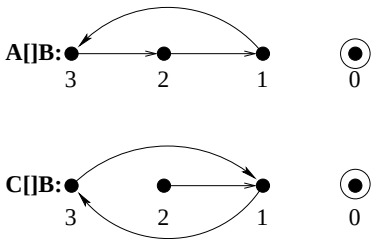


Figure 5.3: $(C \,[]\, B)$ is not a convergence refinement of $(A \,[]\, B)$

As seen in Figure 5.3, $a = 3-2-1-3-2-1-3-2-1...$ is a computation of $A \,[]\, B$, and $c = 3-1-3-1-3-1...$ is a computation of $C \,[]\, B$. However, c drops infinitely many states from a and thus $C \,[]\, B$ is not a convergence refinement of $A \,[]\, B$.

(End of example).

Therefore, in order to prove a composition theorem we need to prove loop-free behavior of the abstract composition outside its invariant.

Lemma 5.12 .

$$[C \preceq A] \ \wedge \ (A \,[]\, B \,[]\, I) \textit{ is stabilizing to } A$$
$$\Rightarrow \quad [C \,[]\, B \,[]\, I \ \preceq \ A \,[]\, B \,[]\, I]$$

Proof: The proof consists of two parts. We prove in the first part that $[C \,[]\, B \,[]\, I \subseteq A \,[]\, B \,[]\, I]_{init}$ and in the second part that every computation of $C \,[]\, B \,[]\, I$ is a convergence isomorphism of some computation of $A \,[]\, B \,[]\, I$.

1. $[C \preceq A] \Rightarrow [C \subseteq A]_{init}$. Thus, every computation of C starting from an initial state is a computation of A and, hence, $[C \,[]\, B \,[]\, I \subseteq A \,[]\, B \,[]\, I]_{init}$.

2. Any computation x of $(C \,[]\, B \,[]\, I)$ can be written as $\cdots - CS_i - BS_i - IS_i - CS_{i+1} - BS_{i+1} - IS_{i+1} \cdots$ where CS denotes consecutive states (potentially empty) produced by C, BS that of B, and IS that of I. Since $[C \preceq A]$, C can only drop states from computations of A. Thus, there exists a computation x' of $(A \,[]\, B \,[]\, I)$ of the form $\cdots - AS_i - BS_i - IS_i - AS_{i+1} - BS_{i+1} - IS_{i+1} \cdots$ where for all i, CS_i is a convergence isomorphism of AS_i.

   Since $(A \,[]\, B \,[]\, I)$ is stabilizing to A (under weak fairness), there are no loops in the uninitialized computations of $(A \,[]\, B \,[]\, I)$. That is, a sequence of states that are outside the invariant of A and occured in AS_i cannot occur again later in AS_j where $j > i$. Thus, in every computation of $C \,[]\, B \,[]\, I$, C can drop only a finite number of states from the corresponding computation of A. Therefore, x can drop only a finite number of states from x', and hence we conclude that x is a convergence isomorphism of x'. □

Lemma 5.13 .

$$[D \preceq B] \ \wedge \ (A \ \Box \ B \ \Box \ I) \textit{ is stabilizing to } A$$
$$\Rightarrow \quad [A \ \Box \ D \ \Box \ I \ \preceq \ A \ \Box \ B \ \Box \ I]$$

Proof: Similar to the proof of Lemma 5.12 . □

Theorem 5.14 *(Composition of convergence refinements).*

$$[C \preceq A] \ \wedge \ [D \preceq B] \ \wedge \ [I' \preceq I]$$
$$\wedge \ (A \ \Box \ B \ \Box \ I) \textit{ is stabilizing to } A$$
$$\Rightarrow \quad [C \ \Box \ B \ \Box \ I \ \preceq \ A \ \Box \ B \ \Box \ I]$$

Proof: Follows from Lemma 5.12 , Lemma 5.13 (applied twice), and transitivity of [$\preceq$]. □

Our composition theorem is very general and is defined in an asymmetric manner. For B to be a stand-alone component rather than a tolerance wrapper, in the abstract one should also prove that $(A \ \Box \ B \ \Box \ I)$ is stabilizing to invariant of B. This way we ensure that starting from the initialized states, both component do useful work.

5.4.4 Refinement between different state spaces

The definitions and theorems introduced in this chapter assumed for the sake of convenience that C and A use the same state space. However, as the examples presented in the introduction illustrate, the state space of the implementation can be different than that of the specification since the implementations often introduce some components of states that are not used by the specifications.

This is handled by relating the states of the concrete implementation with the abstract specification via an abstraction function. The abstraction function is a *total*

mapping from Σ_C, the state space of the implementation C, *onto* Σ_A, the state space of the specification A. That is, every state in C is mapped to a state in A, and correspondingly, every state in A is an image of some state in C.

All definitions and theorems in this chapter are readily extended with respect to the abstraction function. The soundness and completeness of our abstraction functions are discussed in detail in Section 7.1.

5.5 Chapter Summary

In this chapter, we investigated the specification-based design of system stabilization, which uses only the system specification, towards overcoming drawbacks of the tradional whitebox approach, which uses the system implementation as well. The specification-based design approach offers the potential of adding stabilization in a scalable manner, since specifications grow more slowly than implementations. It also offers the potential of component reuse: component technologies typically separate the notion of specification (variously called interface or type) from that of implementation. Since reuse occurs more often at the specification level than the implementation level, specification-based design of stabilization is more reusable than stabilization that is particular to an implementation.

Although we have limited our discussion of the specification-based design approach to the property of stabilization, the approach is applicable for the design of other dependability properties, for example, masking fault-tolerance and fail-safe fault-tolerance. (A system is masking fault-tolerant iff its computations in the presence of the faults implement the specification. A component is fail-safe fault-tolerant

iff its computations in the presence of faults implement the "safety" part [but not necessarily the "liveness" part] of its specification.) Our observation that specification-based design of stabilization is not readily achieved for all specifications is likewise true for specification-based design of masking and specification-based design of fail-safe. Moreover, our observation that local everywhere specifications are amenable to specification-based design of stabilization is also true for specification-based design of masking and specification-based design of fail-safe.

CHAPTER 6

SPECIFICATION-BASED DESIGN OF HEALING FOR TRACKING IN SENSOR NETWORKS

6.1 Introduction

In Chapter 2 we presented a fault-locally self-healing tracking service, STALK, for sensor networks. There, we used I/O automata specification language for describing STALK, and gave formal proofs of correctness and fault-local self-healing for this I/O language program. The implementation languages for sensor network platforms are, however, more finer-grained than the abstract I/O language. For the mote [102] platform, the implementation language is a dialect of C, called NesC [43], and the runtime environment TinyOS [55] consists of a collection of system components for network protocols and sensor drivers. With a conservative estimate, the 20 lines of I/O code we wrote for STALK will correspond to 2000 lines of code (including the libraries for networking and sensing) at the implementation level. Even though we formally verified STALK at the I/O language level, proving correctness and self-healing of the corresponding implementation at the TinyOS level by studying 2000 lines of code is a very challenging task.

In this chapter we revisit STALK and consider the design of specification-based self-healing to the STALK program in order to achieve a self-healing implementation of this tracking service at the TinyOS level. By doing so, we also illustrate that specification-based design of self-healing enables scalability of the design effort of self-healing with respect to the size of the implementation.

In Chapter 5, to enable a specification-based design of stabilization, we presented two fault-tolerance preserving and compositional refinements, namely everywhere and convergence refinements. We can consider using these refinements for implementing STALK in NesC, however, these refinements do not have tool/compiler support and, hence, their adoption in practice is limited. In this case, it would be hard to prove manually that our implementation at the TinyOS level is in fact an everywhere or convergence refinement of STALK at the I/O automata level. Instead, in this chapter, we show that we can use ordinary refinements (for which a lot of tool/compiler support exists) and still achieve a specification-based design of stabilization under suitable conditions.

The obstacles for adopting ordinary refinements (compilers, code transformers, etc.) for specification-based design of self-healing are that ordinary refinements do not preserve fault-tolerance and that ordinary refinements do not satisfy the compositionality property mentioned in Chapter 5: Even though the abstract system composed of the fault-intolerant tracking program A and the self-healing wrapper W is self-stabilizing, when A and W are refined into C and W' at the implementation level, the concrete system might not be stabilizing since starting from faulty states C may interfere with and invalidate the recovery strategy of W'. Even though we proved that starting from faulty states A does not interfere with W, since ordinary

refinements are concerned only with computations starting from good states, computations of C that start from faulty states are unconstrained and may potentially be interfering with W'.

A straightforward way to prevent the interferences between the wrapper and the application code outside the good states is to use atomic wrappers at both the abstract and the concrete systems. When an atomic wrapper is executed it corrects the application to a good state in a single step, and the application code does not have the opportunity to interfere with the execution and the recovery strategy of the wrapper. Similarly, we also require that the wrapper self-stabilizes atomically in order to prevent the application to interfere with the self-stabilization of the wrapper when starting from a faulty state for the wrapper.

Atomic wrappers are, however, infeasible for distributed systems because, in a distributed system, global system state is not available for instantaneous access. Therefore, for effective specification-based design of stabilization of distributed systems, we restrict our attention to wrappers local to each process of the distributed system. At the abstract level, $A = (\Box\ i ::\ A_i)$, we design the wrappers to be decomposable as local wrappers, one for every process i; i.e., $W = (\Box\ i ::\ W_i)$. While refining to a distributed implementation $C = (\Box\ i ::\ C_i)$ we refine these local and atomic wrappers to be composed with the application code C_i at each process i; i.e., $W' = (\Box\ i ::\ W'_i)$

By using local and atomic wrappers we achieve stabilization for each process both at the abstract and concrete system levels. However, even though all the processes are individually stabilizing, a system may fail to stabilize as a whole due to the continuous introduction of corruptions to the system by the processes that are in a faulty state at the time. Consider a scenario where process j is not yet stabilized but i is. If they

interact, i may receive bad input from j, and its state may become bad. Next, when j is corrected to a good state, since i is not yet stabilized, i can in turn infect j. This cycle may repeat infinitely, and even though i and j are individually stabilizing, the system may fail to stabilize as a whole.

In order to ensure that stabilization of individual processes leads to stabilization of the system as a whole, we borrow ideas from literature on compositional approaches to stabilization. One simple idea is stabilization through composition of layers [36]. In the traditional stabilization by layers approach lower-level processes are oblivious to the existence of higher-level processes, and higher-level processes can read (but not write) the state of a lower- level process. Processes can corrupt each other, but only in a predetermined controlled way since lower-level processes cannot be affected by the state of higher-level ones. Also, the order in which correction must take place is the same direction, the correction of higher-levels depend on that of lower-levels. In order to ensure that stabilization compose at the system level, we adopt a layered composition technique at the abstract system level and assert that the concrete system preserve the layered composition structure of the abstract system.

To recap, we make the following assumptions:

1. Wrappers are local to each process and are atomic.

2. Identical layered composition structure is used by the abstract and concrete systems.

These two assumptions are satisfied by a rich class of implementations. For example, STALK satisfies both assumptions. The correctors (wrappers) in STALK are

local to each process and atomic: the process is atomically put into a locally consistent state with respect to the processes it interacts. Also STALK algorithm imposes a static structure on the information flow. There is no communication from a higher level process to a lower level process. The direction of communication is from lower level processes to higher level processes. Due to this structural constraint, the same layered composition structure is applicable at both the abstract and concrete systems.

We show, using these two assumptions, that an ordinary refinement suffices for the fault-intolerant tracking algorithm, and a self-stabilization preserving refinement suffices for the wrappers. The reason we use a stabilization-preserving refinement (e.g., everywhere or convergence refinements) for the wrapper is to ensure that the concrete wrapper is able to stabilize from the corruption of its variables. Since there are a lot of tool support for ordinary refinements, refinement of the tracking algorithm can be done automatically via a compiler. Since the wrappers are small and simple their proof of self-stabilization can be achieved easily even at the implementation level. Pattern-based design of self-healing and automated synthesis of specification-based self-healing approaches, that we discuss in Chapter 9, are also of help for the stabilization preserving refinement of the wrappers.

Outline of the rest of the chapter. In the next section, we show that the refinement method we described above is amenable for the specification-based design approach. We discuss the refinement of STALK to a fault-locally self-healing implementation in Section 6.3. In Section 6.4 we discuss possible extensions to our refinement method by relaxing the layered composition assumption. Finally we conclude the chapter with a summary in Section 6.5.

In this chapter we restrict our presentation to the refinement of the **Tracker** automata of STALK and do not discuss the refinement of the simpler and less interesting **Finder** automata. We note that our method is also applicable for the refinement of the **Finder** automata.

6.2 Adopting Ordinary Refinements for Specification-based Design

We use the same system model as in Section 5.2.

Definition. A *system* S is a finite-state automaton $(\Sigma,\ T,\ I)$ where T, the set of transitions, is a subset of $\{(s_0, s_1) : s_0, s_1 \in \Sigma\}$ and I, the set of initial states, is a subset of the state space Σ.

A computation of S is a maximal sequence of states such that every state is related to the subsequent one with a transition in T, i.e., if a computation is finite there are no transitions in T that start at the final state. We define a wrapper to be a system over Σ and formulate the "addition" of one system to another in terms of the operator $\Box$ (pronounced "box") which denotes the union of automata.

Definition. C is a *refinement* of A, denoted $[C \subseteq A]_{init}$, iff every computation of C that starts from an initial state is a computation of A.

Definition. *C is stabilizing to A* iff every computation of C has a suffix that is a suffix of some computation of A that starts at an initial state of A.

Let A and C be distributed systems composed of processes A_i and C_i respectively; i.e., $A = (\Box\ i ::\ A_i)$ and $C = (\Box\ i ::\ C_i)$. We say that a wrapper W_i for each process A_i is local and atomic iff W_i when executed self-stabilizes (if its state is corrupted) and corrects A_i to a good state (locally consistent state) in a single step.

Stabilization through composition of layers require that the correction and the corruption relations are to the same direction and form a directed acyclic graph. *Corruption relation* denotes for each process in a bad state which other processes it can corrupt. That is the corruption relation constrains the processes an uncorrected process can potentially corrupt. *Correction relation* denotes for each process the prior correction of which other processes its correction depends on. That is, the correction relation constrains the order in which correction must occur.

Theorem 6.1 states the conditions under which ordinary refinements are usable for the specification-based design of stabilization: Given local and atomic wrappers (premise 3) that achieve stabilization of the abstract system (premise 1), ordinary refinement of the application code at each process (premise 2) when composed with everywhere refinement of the abstract wrapper (premise 4) —provided that the layered composition structure of the abstract is preserved (premise 5)— results into a concrete system that is self-stabilizing to the abstract system specifications.

Theorem 6.1 . If

1. $(\Box i :: [A_i \ \Box \ W_i])$ is stabilizing to A_i,

2. $(\forall i :: [C_i \ \subseteq \ A_i]_{init})$,

3. $(\forall i :: [W_i' \ \subseteq \ W_i])$,

4. $(\forall i :: W_i$ is local and atomic), and

5. the correction & corruption relations of the abstract system are to the same direction, and the concrete system preserves the correction & corruption relations of the abstract

then $(\Box i :: [C_i \ \Box \ W_i])$ is stabilizing to $(\Box i :: A_i)$.

Proof. From premises 3 and 4 it follows that $(\forall i :: W_i'$ is local and atomic), and hence C_i cannot interfere with the recovery strategy of W_i'. Thus, from premises 2, 3, and 4, it follows that $(\forall i :: [C_i \ \Box \ W_i]$ is stabilizing to $[A_i \ \Box \ W_i])$. The conclusion follows from this result and premises 1 and 5. □

6.3 Refinement of STALK to the Implementation Level

In this section, we present a refinement of the abstract STALK program given in Chapter 2 to the TinyOS implementation level by showing that Theorem 6.1 is applicable for this refinement. We start by recalling some of the properties of STALK and pointing out which concepts of Theorem 6.1 they correspond to. We then continue with a discussion of the refinement to the implementation level.

STALK provides local specifications for the fault-intolerant tracking program: The $\mathbf{Tracker}_i$ automata presented in Section 2.4 corresponds to A_i in Theorem 6.2.

STALK also provides local and atomic wrappers for each $\mathbf{Tracker}_i$: The parallel composition of the correction actions in Section 2.5 corresponds to W_i in Theorem 6.2. Since, in Chapter 2, we proved that $\mathbf{Tracker}_i$ composed with the correction actions are fault-locally self-stabilizing, premise 1 is satisfied. Since the correction actions for the $\mathbf{Tracker}_i$ automata are all local and atomic (since they are stateless, they are stabilizing in one step; they also put $\mathbf{Tracker}_i$ in a locally-consistent state in one step), premise 4 is satisfied.

STALK imposes a static layered structure on the processes: There is no communication from a higher level process to a lower level process; the direction of communication is from lower level processes to higher level processes. Due to this structural constraint, the same layered composition structure is applicable at both the abstract and concrete systems; hence, premise 5 is satisfied.

Next, we consider the refinement of STALK to the implementation level. In order for Theorem 6.1 to be applicable, we need to show that premises 2 and 3 are satisfied by our refinement of STALK.

Premise 2 asserts that the implementation of the $\mathbf{Tracker}_i$ automata should be a refinement from the initial states. Since there are a lot of tool support for ordinary refinements, refinement of the tracking algorithm can be done automatically via a compiler. For example, the IOA toolkit [41] supports the design, analysis, verification, and refinement of programs written in I/O automata notation. The toolkit includes analysis tools such as the IOA simulator [61] and interfaces to theorem-proving tools [40] as well as compilers for generation of distributed code in commercial programming languages [101]. Even if the implementation of $\mathbf{Tracker}_i$ automata is performed manually, the verification process for ordinary refinements are, in general, easier than

that of fault-tolerance preserving and compositional refinements. Since an ordinary refinement from initial states of the $\mathbf{Tracker}_i$ automata is sufficient, one does not have to consider refinements from every state for the purposes of this implementation.

Premise 3 asserts that the abstract wrappers should be everywhere refined. IOA toolkit can again be employed to this end. First, using the IOA toolkit we obtain a wrapper at the implementation level, and then we modify this concrete wrapper until we can prove that it is an everywhere refinement of the abstract wrapper. [108] suggests a method for reasoning about the relationships between designs at different levels of abstraction using the IOA toolkit. The method advocates paired execution of the abstract level and concrete level systems using IOA simulator [61] in order to identify an abstraction function (simulation relation in their model). Then, using the Larch theorem prover [40] the refinement is formally verified, potentially from every state. Alternatively, model-checking based approaches may be used for the verification: For example, [49] can accept a wrapper written in C language as input, and model-check properties of the wrapper. Since the wrappers are small and simple, their proof of self-stabilization can be achieved easily even manually, without any tool support. Since sensor nodes [54] have a single thread of control, the concrete level wrappers can also be made atomic easily. Also, since stateless wrappers are trivially self-stabilizing, self-stabilization of the wrapper can be achieved by avoiding introduction of new variables or by using soft-state variables.

Since all the premises are satisfied, we can conclude, by a simple application of Theorem 6.1 , that the resultant implementation of STALK at the TinyOS level is self-stabilizing to the abstract specifications.

Even though preservation of stabilization is guaranteed by our refinement method, preservation of fault-containment demands a stronger refinement for the wrappers: aside stabilization property, real-time guarantees of the abstract wrappers should also be preserved. The fault-containment calculations in STALK made use of the detection and execution time by the local wrappers: The time latency of detection at the "start-shrink" action depends on the timeout period of the heartbeats, and the execution time of the wrappers are assumed negligible. When we refine the wrappers, we should be careful about the detection and execution latencies at the concrete level. If the detection and execution latencies introduced at the concrete level do not influence fault-containment calculations, we can reuse the fault-containment argument of STALK at the concrete system level. By ensuring that the real-time requirements of STALK are respected by the implementation, we can conclude that the fault-containment property of STALK is respected by the concrete system.

6.4 Extensions

In this section, we discuss possible extensions to our refinement method by relaxing the layered composition assumption, that we adopted for achieving specification-based design of self-healing using ordinary refinements.

In [73], a compositional framework for constructing self-stabilizing systems is proposed. The framework explicitly identifies for each component which other components it can corrupt (corruption relation). Additionally, the correction of one component often depends on the prior correction of one or more other components, constraining the order in which correction can take place (correction relation). Depending on what is actually known about the corruption and correction relations, the

framework offers several ways to coordinate system correction. In cases where the correction and corruption relations are in reverse directions, persistent corruption cycles may be formed: even though all the components are individually stabilizing, the system may fail to stabilize due to the continuous introduction of corruptions to the system via these corruption cycles. By employing blocking coordinators, the framework breaks these malicious cycles. In cases where both correction and corruption relations are in the same direction, no cycle forms and there is no need for blocking.

By including both correction and corruption relations, this framework subsumes and extends other compositional approaches, such as layered composition, where correction and corruption relations are to the same direction. By adopting this framework in our refinement method, we can relax our layered composition assumption and allow arbitrary compositions of processes. Using the knowledge of correction-corruption relations between the processes, we can instantiate a corresponding coordinator to ensure that stabilization of processes compose at the system level. We can even relax the preservation of correction corruption relations when going to the concrete if we know the correct coordinator to use for the correction-corruption relations at the concrete.

6.5 Chapter Summary

In this chapter we revisited the tracking problem in sensor networks and presented design of specification-based self-healing to STALK in order to achieve scalability of design effort of self-healing with respect to the implementation of this tracking service.

More specifically, we showed that we can use ordinary refinements (for which a lot of tool/compiler support exists) and still achieve a specification-based design of

stabilization under suitable conditions. To this end, we assumed that (1) wrappers are local to each process and are atomic, and (2) the concrete system preserves the layered composition structure of the abstract system.

Using these two conditions, we showed that ordinary refinement suffices for the fault-intolerant tracking algorithm, and a self-stabilization preserving refinement suffices for the wrappers. Since there are a lot of tool support for ordinary refinements, refinement of the tracking algorithm can be done automatically via a compiler.

CHAPTER 7

DISCUSSION ON ABSTRACTION FUNCTIONS AND AUTOMATED SYNTHESIS OF SPECIFICATION-BASED DESIGN OF FAULT-TOLERANCE

In this chapter, we first investigate in Section 7.1, the soundness and completeness of the abstraction functions we use in our specification-based design approach, and then, in Section 7.2, we discuss a preliminary method for achieving automated synthesis of specification-based design of fault-tolerance.

7.1 Soundness and Completeness of Abstraction Functions

In this section, we investigate soundness and completeness of our abstraction functions. To this end, we first summarize our use of abstraction functions briefly in Section 7.1.1. Then in 7.1.2, we present two constructive methods that given a system C deduces an abstract system A such that $[C \subseteq A]$ and $[C \preceq A]$, respectively, and prove the soundness and completeness of these abstraction methods. Finally, in Section 7.1.3 we present a survey of abstraction functions developed in the model checking literature that are applicable for specification-based design.

7.1.1 Abstraction functions

In this section, after some preliminary definitions, we justify how abstraction functions allow the implementation to have a different state space than that of specification.

Let Σ be a state space.

Definition. A *system* S is a finite-state automaton (Σ, T, I) where T, the set of transitions, is a subset of $\{(s0, s1) : s0, s1 \in \Sigma\}$ and I, the set of initial states, is a subset of Σ.

A computation of S is a maximal sequence of states such that every state is related to the subsequent one with a transition in T, i.e., if a computation is finite there are no transitions in T that start at the final state.

Abstraction between different state spaces. The state space of the implementation C can be different than that of the specification A since the implementations often introduce some components of states that are not used by the specifications. We handle this by relating the states of the concrete implementation with the abstract specification via an abstraction function. The abstraction function is a *total* mapping from Σ_C, the state space of the implementation C, *onto* Σ_A, the state space of the specification A. That is, every state in C is mapped to a state in A, and correspondingly, every state in A is an image of some state in C.

Additional state variables in the concrete. Note that our abstraction function allows C to introduce irrelevant variables for implementing a feature that is orthogonal to the functionality of A (e.g., a graphical user interface at C). If only the irrelevant variables differ for two states c_1 and c_2 of C, then our abstraction functions will map c_1 and c_2 to correspond the same state in A.

7.1.2 Soundness and completeness of abstraction functions

Everywhere refinements. We present a constructive method that given a system C deduces an abstract system A such that $[C \subseteq A]$. This method is equivalent to the existential abstraction method presented in model-checking literature in Section 7.1.3.

Theorem 7.1 *(Soundness).* Let h be an abstraction function from C to A. If

1. $I_A(s_A) \iff (\forall s_C : h(s_C) = s_A : I_C(s_C))$

2. $T_A(s0_A, s1_A) \iff (\exists s0_C, s1_C : h(s0_C) = s0_A \ \wedge \ h(s1_C) = s1_A : T_C(s0_C, s1_C))$

then $[C \subseteq A]$. (That is, every computation of C is a computation of A with respect to the abstraction function h.) □

Theorem 7.2 *(Completeness).* If $[C \subseteq A]$

then there exists an abstraction function h from C to A such that

1. $I_A(s_A) \iff (\forall s_C : h(s_C) = s_A : I_C(s_C))$

2. $T_A(s0_A, s1_A) \iff (\exists s0_C, s1_C : h(s0_C) = s0_A \ \wedge \ h(s1_C) = s1_A : T_C(s0_C, s1_C))$

Proof. $[C \subseteq A]$ implies that every computation of C is a computation of A with respect to some abstraction function h. This same abstraction function h satisfies the two conditions in the antecedent of the theorem, therefore such an existential abstraction exists. □

Convergence refinements. We present a constructive method, "*expansion abstraction*", that given a system C deduces an abstract system A such that $[C \preceq A]$. Recall that convergence refinement implies that even in the unreachable states the computations of the concrete system C track the computations of the abstract system

A, although some states that appear in the computations of A may disappear in the computations of C, and hence, C preserves convergence properties (e.g., stabilization) of A.

The idea of expansion abstraction is similar to the existential abstraction method we presented above. We first construct A as usual using the existential abstraction method we described above. Then, we investigate the transitions of A outside the invariant states of A and expand some of these transitions: If there is a transition from state s_0 to s_N and there exists another sequence of transitions starting from s_x and ending at s_N, then we remove the transition from s_0 to s_N and add a transition from s_0 to s_x.

Theorem 7.3 *(Soundness).* Let h be an abstraction function from C to A. If there exists an expansion abstraction from C to A that uses h, then $[C \preceq A]$. (That is, $[C \subseteq A]_{init}$ and every computation of C is a compression of a computation of A with respect to the abstraction function h.) □

Theorem 7.4 *(Completeness).* If $[C \preceq A]$,

then there exists an expansion abstraction from C to A.

Proof. $[C \preceq A]$ implies that there exists an abstraction function h such that $[C \subseteq A]_{init}$ with respect to h and every computation of C is a compression of a computation of A with respect to h. This implies that there exists an expansion abstraction from C to A that uses h. □

7.1.3 Abstraction functions in model checking literature

Model checking is an automatic verification technique for finite state concurrent systems. Typically, the user provides a high level model of the system and the temporal logic formula to be checked. The formula is then automatically checked by an exhaustive search of the state space of the model. The model checking algorithm will either terminate with the answer true, indicating that the model, and hence the system, satisfies the formula, or give a counterexample execution showing why the formula is not satisfied.

The main challenge in model checking is the state explosion problem. Abstraction is an effective technique for tackling this problem. Abstraction has been traditionally a manual process requiring creativity, however, recently some abstraction techniques [26, 27, 29, 31] are proposed to automate this process entirely.

In this section we briefly summarize an automated abstraction technique proposed by Clarke et al. [26, 27] and hint on its application to automated synthesis of specification-based fault-tolerance. We leave the details and demonstration of these techniques to Section 7.2, where we demonstrate our automated synthesis design of specification-based fault-tolerance.

In [26, 27] model checking of ACTL formulas are considered. ACTL is the fragment of CTL where only the operators involving A ("for every path") are used, and negation is restricted to atomic formulas. ACTL is expressive enough to capture stabilizing, masking, and fail-safe fault-tolerance properties. There Clarke et al. introduce the concept of *existential abstraction*. An abstraction function h is described by a surjection (i.e., h is onto) $h : S \longrightarrow S'$ where S is the set of concrete states and

S' the set of abstract states. In contrast, in our work [32] we have defined an abstraction function to be total and onto from S_C to S_A. We have required the abstraction function to be total since we also consider faulty computations of the concrete and do not restrict ourselves to merely the initialized computations.

Given a concrete system $C = (S, I, R)$ (where I denotes the initial states of C and R denotes a total transition relation on S) the abstract system $A = (S', I', R')$ corresponding to the abstraction function h is defined as follows:

1. $I'(d')$ iff $\exists d(h(d) = d' \ \wedge \ I(d))$
2. $R'(d'_1, d'_2)$ iff $\exists d_1 \exists d_2(h(d_1) = d'_1 \ \wedge \ h(d_2) = d'_2 \ \wedge \ R(d_1, d_2))$

Existential abstraction guarantees that when an ACTL formula is true in A, it will also be true in C. However, since A contains less information, model checking A potentially leads to wrong results: If a formula is false in A, this may be the result of some behavior in the approximation that is not present in C.

Consider the following spurious counterexample.

Example: Assume that for a traffic light controller we want to prove $\phi =$ **AGAF***(state = red)* (i.e., red state is encountered infinitely often on every computation path) using the abstraction function *h(red) = red* and *h(green) = h(yellow) = go*. From the figure it is easy to see that while the concrete system C satisfies ϕ, the abstract system A does not: there exists an infinite trace "*red, go, go, go, ...*". Since this abstract counterexample does not correspond to some concrete counterexample, it is a spurious counterexample [26].

(End of example)

Existential abstraction produces a loose approximation of the concrete system and hence leads to spurious counterexamples. On the other hand, it is weak enough to

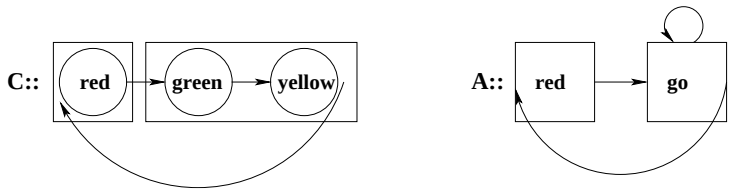


Figure 7.1: Spurious counterexample

be achieved by automated methods (see Section 4.2 of [28]). Next we present the automatic abstraction-refinement methodology in [26] in which the initial abstract model is generated by an automatic analysis of the control structures in the program to be verified.

(Remark): In our work on everywhere refinements [6] we use a stronger notion of abstraction. Instead of the $\exists d_1 \exists d_2$ in condition 2 of existential abstraction method, this more strict method uses $\forall d_1 \exists d_2$ quantification. Thus, this method avoids the spurious counterexample problem. In [28], another abstraction "exact-approximation", that uses $\forall d_1 \forall d_2$ quantification for condition 2, is introduced. Using exact-approximation, a property will be true at the concrete level if and only if it is true at the abstract level. Exact-approximations are too strict to be useful: the authors [28] report that exact approximation allows very little simplification, and hence not useful for reducing complexity of verification. Our abstraction function in [6] is not as strict as exact-approximations, but still prevents the existence of spurious counterexamples. *(End of remark)*

Counterexample-guided abstraction:

1. *Generate the initial abstraction h automatically*: First, by examining the transition blocks corresponding to the variables of the program C, *formula clusters*

are formed: A variable cannot appear in formulas that belong to two different formula clusters. Then, an existential abstraction [28] on the value domains of the variables in a formula cluster are defined such that two values are in the same equivalence class if they cannot be distinguished by atomic formulas appearing in that formula cluster.

2. *Model-check the abstract structure*: Let A be the abstract system (Kripke structure) corresponding to the abstraction h. Standard model checking techniques are used to see whether A satisfies the given ACTL formula. If the check is confirmative, then we conclude that C also satisfies that formula. Else, if the check reveals a counterexample, we check whether this is also a counterexample in C. If it is an actual counterexample it is revealed to the user, else if it is a spurious counterexample, we proceed to step 3.

3. *Refine the abstraction*: The equivalence class in A that has caused the spurious counterexample is determined and is partitioned so that the refined abstraction function h no longer admits that spurious counterexample.

We give an example of the above procedure in Section 7.2, where we discuss application of automatic generation of abstraction functions in automated synthesis of specification-based fault-tolerance. The idea, briefly, is that given a fault-intolerant concrete system C, we will deduce a corresponding abstract system A using the automatic abstraction methodology discussed in this section. We will then automatically synthesize a tolerance wrapper for achieving fault-tolerance of A and refine this wrapper (using the abstraction in the reverse direction) to provide fault-tolerance to the concrete system C. Note that the wrapper synthesized for the abstract model A is readily available for mapping back to C because the abstraction function is defined

as onto. In fact in [26] a method for mapping the counterexamples in the abstract model to counterexamples in the concrete model is presented. We simply adopt that method to map the abstract tolerance wrapper to the concrete system level.

Our motivation for automated synthesis for specification-based fault-tolerance is the same as that of using abstractions in model checking: to avoid the state explosion problem. Since A will be an abstraction of C, our specification-based automated fault-tolerance synthesis algorithm will be scalable and low-cost.

Existential abstractions are insufficient for convergence refinements: The above abstraction technique, given a concrete system C will produce (after some iterative refinement of the abstraction function) a corresponding abstract system A such that $[C \subseteq A]$ (i.e., C is an everywhere refinement of A). Even though $[C \subseteq A] \Rightarrow [C \preceq A]$, the existential abstraction functions cannot truly capture the expressive power of convergence refinements. We present one such abstraction in Section 7.1.

Model-checking with well-founded bisimulation. In [80], Manolios et al. propose an approach to verification that combines the strengths of the model checking and the automated theorem proving approaches: They use a theorem prover to reduce an infinite-state system to a finite-state system, which they then handle using automatic methods.

The reduction amounts to proving a stuttering bisimulation that preserves the properties of interest. To this end, they introduce the concept of well-founded equivalence bisimulation. As a demonstration of their approach they verify the alternating bit protocol.

This approach might be useful for automated synthesis of specification-based fault-tolerance to infinite-state systems. However, they essentially use an abstraction function similar to existential abstraction to deduce an abstract system given a finite-state concrete system that is a well-founded equivalence bisimulation of an infinite-state concrete system.

7.2 Automated Synthesis of Specification-Based Tolerance

In this section we demonstrate our automated synthesis of specification-based fault-tolerance by designing fault-tolerance to two 3-state token-ring systems. To this end, we modify Clarke et al.'s [26] automated algorithm for finding an existential abstraction function so that when we deduce an abstract system for the given concrete system, instead of model-checking for a tolerance property in the abstract, we run an automated synthesis algorithm on the abstract to find a tolerance wrapper that would ensure that tolerance property.

Recall from Section 7.1.3 that since every computation of the concrete system is a computation of the abstract system, the synthesized abstract tolerance wrapper suffices for achieving the desired level of fault-tolerance at the concrete also. However, if the abstraction is approximate (e.g. existential abstraction) then parts of the abstract wrapper may turn out to be vacuous when the wrapper is mapped to the concrete level. This phenomenon is analogous to the spurious counterexample concept in [26]. We have presented such an example in 4-state token-ring systems in [32].

In our work on everywhere refinements [6] we have used a stronger notion of abstraction. Instead of the $\exists d_1 \exists d_2$ in condition 2 of existential abstraction method, we have used $\forall d_1 \exists d_2$ quantification. Using such an abstraction, we are guaranteed

that (1) every computation of C is a computation of A, and (2) every computation of A is a computation of C. Therefore, if there is a way to wrap the concrete system C to achieve a desired level of fault-tolerance, we are guaranteed to find a suitable abstract tolerance wrapper for A to achieve that desired level of fault-tolerance.

(Remark): The existing work on synthesis of fault-tolerance [5, 63] cannot synthesize a separate tolerance component but instead output a fault-tolerant transformation of the input system. Thus, here we will derive the tolerance wrappers by ourselves and not by using an automatic synthesis procedure. *(End of remark)*

(Remark): The abstract systems we produce in this section are such that every computation of the concrete system is a computation of the abstract system. In another words $[C \subseteq A]$. Had we given another abstract system A' such that $[C \preceq A']$ (i.e., C is a convergence refinement of A), our method would still apply. That is, once a tolerance wrapper is synthesized for A, mapping it to the concrete level would still achieve stabilization of C. *(End of remark)*

7.2.1 BTR1: A fault-intolerant token-ring system

BTR1 consists of 3 processes.

$$
\begin{array}{lcl}
c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 \\
c.0 = c.1 \oplus 1 & \longrightarrow & c.1 := c.0 \\
c.2 = c.1 \oplus 1 & \longrightarrow & c.1 := c.2 \\
c.1 = c.2 \oplus 1 & \longrightarrow & c.2 := c.1 \oplus 1
\end{array}
$$

Finding the abstraction function. Every process j maintains a counter $c.j$ with the domain $\{0, 1, 2\}$. The set of atomic formulas in BTR1 is $\{(c.1 = c.0 \oplus 1), (c.0 = c.1 \oplus 1), (c.2 = c.1 \oplus 1), (c.1 = c.2 \oplus 1)\}$. Two atomic formulas f1 and f2 interfere

iff "$var(f1) \cap var(f2) \neq 0$". The formulas interfering with each other should be put into the same formula cluster, thus, all the atomic formulas in BTR1 are in the same formula cluster. Therefore, variables $c.0$, $c.1$, and $c.2$ are in the same variable cluster.

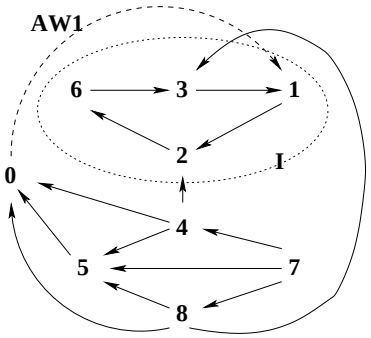
We construct the initial abstraction function h as follows. We put two states in the same equivalence class if the values of variables in the two states cannot be distinguished by atomic formulas appearing in that cluster. The domain of $\{0,1,2\} * \{0,1,2\} * \{0,1,2\}$ is partitioned into a total of 9 equivalence classes by this criterion. We denote these classes by the natural numbers 0...8.

$$
\begin{array}{rcl}
0 & = & \{(0,0,0),\ (1,1,1),\ (2,2,2)\} \\
1 & = & \{(0,0,2),\ (1,1,0),\ (2,2,1)\} \\
2 & = & \{(0,0,1),\ (1,1,2),\ (2,2,0)\} \\
3 & = & \{(0,2,2),\ (1,0,0),\ (2,1,1)\} \\
4 & = & \{(0,2,1),\ (1,0,2),\ (2,1,0)\} \\
5 & = & \{(0,2,0),\ (1,0,1),\ (2,1,2)\} \\
6 & = & \{(0,1,1),\ (1,2,2),\ (2,0,0)\} \\
7 & = & \{(0,1,0),\ (1,2,1),\ (2,0,2)\} \\
8 & = & \{(0,1,2),\ (1,2,0),\ (2,0,1)\}
\end{array}
$$

That is, h maps states $(0,0,0)$, $(1,1,1)$, and $(2,2,2)$ of BTR1 to state 0 of the abstract system ABTR1.

Producing the abstract system, ABTR1. From the definition of existential abstraction and h, BTR1 follows:

Abstract tolerance wrapper, AW1. It is easy to see from the figure that ABTR1 becomes stabilizing if a transition is added from 0 into the invariant states. The abstract wrapper adds a transition from 0 to 1. This is the same as inserting a token at process N when there are no tokens in the system.



Refining the tolerance wrapper into W1. When we map AW1 to the concrete level, we get the following wrapper W1:

$$c.1 = c.2 \ \wedge \ c.2 = c.0 \ \longrightarrow \ c.2 = c.1 \oplus 2$$

From the specification-based design theorem it follows that W1 provides stabilizing fault-tolerance to BTR1. Indeed, BTR1 ▯ W1 results in Dijkstra's 3-state token-ring algorithm.

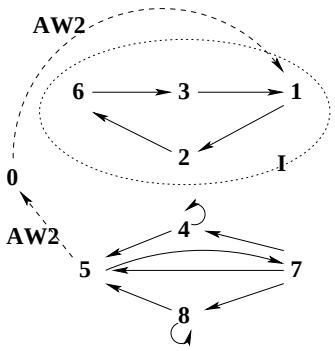
7.2.2 BTR2: Yet another fault-intolerant token-ring system

BTR2 consists of 3 processes.

$$\begin{array}{lcl} c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 \\ c.0 = c.1 \oplus 1 & \longrightarrow & c.1 := c.2 \oplus 1 \\ c.2 = c.1 \oplus 1 & \longrightarrow & c.1 := c.0 \oplus 1 \\ c.1 = c.2 \oplus 1 & \longrightarrow & c.2 := c.1 \oplus 1 \end{array}$$

Since the guards of BTR2 is the same as that of BTR1, the abstraction function remains the same for BTR2 as in BTR1.

Producing the abstract system, ABTR2. From the definition of existential abstraction and h, BTR1 follows:



Abstract tolerance wrapper, AW2. It is easy to see from the figure that ABTR2 becomes stabilizing if (1) a transition is added from 5 to 0, and (2) a transition is added from 0 into the invariant states.

We should give priority to the wrapper so that it executes first at state 5 and eliminates the possible infinite, non-converging computations by breaking the loop.

Note that the loops at 4 and 8 are self-loops and they are taken care of by the weak-fairness assumption: Since a self-loop does not alter the current state, there is always another transition that is infinitely enabled at states 4 and 8, which will eventually be taken.

As before, the transition from 0 to 1 is the same thing as inserting a token at process N when there are no tokens in the system.

Refining the tolerance wrapper into W2. When we map AW2 to the concrete level, we get the following wrapper W2:

$$c.1 = c.2 \ \wedge \ c.2 = c.0 \ \longrightarrow \ c.2 = c.1 \oplus 2$$

$$c.1 \oplus 1 = c.0 \ \wedge \ c.1 \oplus 1 = c.2 \ \longrightarrow \ c.1 = c.0$$

From the specification-based design theorem it follows that W2 provides stabilizing fault-tolerance to BTR2. BTR2 $\square$ W2 results in a slightly different token-ring system than Dijkstra's 3-state token-ring.

CHAPTER 8

RELATED WORK ON SPECIFICATION-BASED DESIGN OF FAULT-TOLERANCE

In this chapter we start with a survey of fault-tolerance design methods in Section 8.1. We then consider related work on fault-tolerance preserving refinements in Section 8.2. Finally, in Section 8.3 we discuss work on scalable design of self-healing through compositional approaches.

8.1 Fault-tolerance Design Methods

In this section we present some previous work on fault-tolerance design methods and compare them with our specification-based design method. To this end, we first give a brief summary our method for specification-based design of fault-tolerance in Section 8.1.1.

We categorize the previous work on fault-tolerance methods with respect to the type of tolerance they provide: We start with methods applicable for design of any type of tolerance in Section 8.1.2, in Section 8.1.3 we survey the methods for designing masking fault-tolerance, in Section 8.1.4 methods for designing nonmasking fault-tolerance, and in Section 8.1.5 methods for fail-safe fault-tolerance. As we study each method, we compare and contrast it with our method.

8.1.1 Brief overview of specification-based design method

Given a high-level system specification A, the specification-based approach is to design a tolerance wrapper W such that adding W to A yields a fault-tolerant system. The goal is to ensure that for any low-level refinement (implementation) C of A adding a low-level refinement W' of W would also yield a fault-tolerant system.

Since the refinements from A to C and W to W' can be done independently, specification-based design enables a posteriori or dynamic addition of fault-tolerance. That is, given a concrete implementation C, it is possible to add fault-tolerance to C as follows:

- First, design an abstract (high-level) tolerance wrapper W using solely an abstract specification A of C, and then
- add a concrete (low-level) refinement W' of W to C.

Note that the goal of specification-based fault-tolerance is not readily achieved for all refinements. The refinements we need for achieving specification-based fault-tolerance should not only preserve fault-tolerance but also have nice composability features so that the refinements from A to C and W to W' can be done independently. In current research, we have presented two such refinements: everywhere refinements [6] and convergence refinements [32]. These refinements ensure that if A composed with W is fault-tolerant, then for any everywhere or convergence refinement C of A adding an everywhere or convergence refinement W' of W would also yield a fault-tolerant system.

8.1.2 Generic methods for fault-tolerance

Method by A. Arora and S. Kulkarni. Arora and Kulkarni [11] have described a comprehensive and widely adopted methodology for the design of tolerance components to a given intolerant system to achieve a desired level of tolerance. In their methodology, they have identified two types of tolerance components, *detectors* and *correctors*, and have demonstrated how these detectors and correctors can be constructed in a hierarchical and efficient manner to render a fault-intolerant system to be fault-tolerant. As illustrations of their methodology, they have designed masking, nonmasking and fail-safe tolerance to several distributed algorithms in the literature [64–67].

Their methodology, however, assumes access to implementation details of the systems they render tolerant. In contrast, in our methodology we achieve design of fault-tolerance using only the system specification and not the system implementation. In fact, in our work we also adopt their methodology while designing fault-tolerance at the abstract system level. The abstract tolerance wrappers we design corresponds to the detector and corrector components in their work.

Method by D. Peled and M. Joseph. In [88], Peled and Joseph view the incorporation of a recovery program into a fault-intolerant program as a *program transformation*. They introduce a proof method which builds on program transformations and corresponding formula transformations. Let T be a program transformation and $\mathcal{T}$ the corresponding formula transformation. The proof rule states that if a property ϕ holds for a program C, then $\mathcal{T}(\phi)$ holds for $T(C)$. This proof rule makes it possible to prove just once that a formula transformation corresponds to a program

transformation, removing the need to prove separately the correctness of each transformed program. The method uses linear temporal logic (LTL) [79] for expressing formulas. One can think of these formulas as specifications of the programs. Then, this proof rule makes it possible to reason about $T(C)$ without looking at its code. For example, if T has some complicated recovery transformation, given that C has a property(i.e., specification) ϕ, apply the transformation $\mathcal{T}$ to ϕ and then conclude using the proof rule that C also satisfies $\mathcal{T}(\phi)$.

This method is limited by the inherent assumption that a fault-tolerant program is obtained by superposing (i.e., layering composition) a generic recovery program (e.g. checkpointing and recovery) to a fault-intolerant program. Hence, fault-tolerant programs that cannot be designed in this manner cannot be specified using their method. In contrast our specification-based design method is not limited with these assumptions.

8.1.3 Design of masking fault-tolerance

Definition. A system is masking tolerant if in the presence of faults, it always satisfies its safety properties and, when faults stop occuring, it eventually resumes satisfying its liveness properties.

Replication. One commonly used method for designing masking fault-tolerant systems is *replication*: the intolerant system is replicated and the output of the replicas is combined to determine the output of the fault-tolerant system. The number of replicas and the way in which the output is combined depends upon the number of faults that need to be tolerated and the types of these faults. For example, if the intolerant program fails in a fail-stop manner, i.e., upon failure, it stops executing and never outputs incorrect values (although it may not output a value), then to tolerate

t faults, the number of replicas required is t+1. As a special case, if t is 1, two replicas are required; this case is an instance of primary-backup. If the intolerant program fails in a Byzantine manner, i.e., upon failure it may output incorrect values, then to tolerate t faults, the number of replicas required is $2t$+1. As a special case, if t is 1, three replicas are required; this case is an instance of triple modulo redundancy.

Replication is a blackbox approach to fault-tolerance, in that it does not exploit any information about the system it renders fault-tolerant, and hence results in systems that have high overhead. Also, if the intolerant system is nondeterministic, replication will not be applicable.

Next we present four other methods for masking tolerance that use the replication idea: state-machine approach, checkpointing & recovery (replication in time rather than space), recovery blocks, and N-version programming. The comments in the paragraph above also applies for these methods.

State-machine approach. Schneider [95] generalizes replication to the client-server model. In his so-called *state-machine approach*, each client and each server is replicated and a communication from a client c to a server s is replaced with a communication from every replica of c to every replica of s. Likewise, a communication from s to c is replaced with a communication from every replica of s to every replica of c. Since each replica of the server (client) gets these messages, each replica of the server (client) can ignore the messages from the erroneous clients.

Checkpointing and recovery. Yet another method is *checkpointing-and-recovery*: after the occurrence of a fault, the program is restored to some previous state in its computation. Towards this end, the intolerant program is augmented

with a checkpointing program and a recovery program; the former records the program state periodically and the latter recovers the program to a recorded good state and resumes the program.

Recovery blocks. Randell [92] combines exceptions and checkpointing-and-recovery. He assumes that a fault is due to an error in the implementation and uses multiple implementations to satisfy a given specification. In particular, a program is composed of *recovery blocks* that consist of an acceptance test and a sequence of subprograms. To execute the recovery block, the first subprogram in this sequence is executed and the acceptance condition is tested thereafter. If the acceptance condition is satisfied, the execution of the recovery block completes correctly. Otherwise, an exception is raised and the program state is recovered to the initial state from where the next subprogram is executed, and so on. Thus, as long as the execution of at least one subprogram satisfies the acceptance condition, the recovery block completes correctly.

N-version programming. N-version programming [13] is similar to recovery blocks idea in that multiple versions of a component are designed to satisfy the same basic requirements. The fundamental difference of this approach from recovery blocks is that N-version requires the use of a generic decision algorithm (a voter) to decide the correct output based on the comparison of all the outputs, whereas recovery blocks approach uses an application dependent acceptance test.

An example of an N-version fault-tolerance design was presented in [93], which uses abstraction to reduce the cost of Byzantine fault tolerance and to improve its

ability to mask software errors. Using abstractions, this work manages to hide implementation details of replicas to enable the reuse of off-the-shelf and, to a certain extent, nondeterministic implementations.

To this end, the authors define a common abstract behavioral specification for the system and implement appropriate conversion functions (i.e., conformance wrappers) for the state, requests, and replies of each implementation in order to make it behave according to a common specification. They also provide a reset mechanism to repair faulty replicas: When a replica is recovered, it is rebooted and restarted from a clean state. Then it is brought up to date using a correct copy of the abstract state that is obtained from the group of replicas.

Since abstractions and abstract states are used for achieving fault-tolerance, this work is close to our work in spirit. However, our method is more general in that we are not limited to only replication-based approaches and masking fault-tolerance.

One of the principal objections to N-version programming is that it is still prone to common mode failure. The objection is based on the observation that programmers tend to think in the same way and therefore even independently written programs are likely to fail in the same way. Also, since multiple versions of a component is used this approach requires considerable development effort.

8.1.4 Design of nonmasking fault-tolerance

Definition. A system is nonmasking tolerant if in the presence of faults, it need not satisfy its safety properties but, when faults stop occuring, it eventually resumes satisfying its safety and liveness properties. A special case of nonmasking tolerance is stabilizing tolerance: in the presence of faults, even if the system reaches an arbitrary state, it eventually recovers to a state from where its safety and liveness properties

are satisfied. In this section, we focus on self-stabilizing fault-tolerance (also known as self-healing), though the same arguments hold for design of nonmasking tolerance.

Research in stabilization [36,39,46,52] has traditionally relied on the availability of a complete system implementation. The standard approach to reasoning uses knowledge of all implementation variables and actions to exhibit an "invariant" condition such that if the system is properly initialized then the invariant is always satisfied and if the system is placed in an arbitrary state then continued execution of the system eventually reaches a state from where the invariant is always satisfied. Likewise, the generic methods for designing stabilization [1, 8, 60, 105] also assume implementation-specific details as input: [8, 60] assume the availability of the implementation invariant, [1] relies on the knowledge of the implementation actions, and [105] takes as input a "locally checkable" consistency predicate derived from implementation.

To the best of our knowledge, our work [6] is the first time that system stabilization is shown to be provable without whitebox knowledge. As one piece of evidence, we offer the following quote due to Varghese [105] (parenthetical comments are ours):

> In fact, the only method we know to *prove* a behavior stabilization result (i.e., stabilization with respect to system specification) is to first prove a corresponding execution stabilization result (i.e., stabilization with respect to system implementation) ...

8.1.5 Design of fail-safe fault-tolerance

Definition. A system is fail-safe tolerant if in the presence of faults, it always satisfies its safety properties but, when faults stop occuring, it need not resume satisfying its liveness properties.

In the literature, there has been some instances [9, 107] of specification-based fail-safe systems prior to our work. This is due to the fact that fail-safe tolerance can be

achieved by using only a "detector" component. It is easy to define an abstraction mapping over a concrete system and use an abstract detector component to detect whether the abstract predicate is truthified. Note that it is not necessary to find a means to refine the detector component to the concrete system level, whereas, that would have been necessary for a corrector component.

Exception handling. One commonly used method for fail-safe fault-tolerant systems is *exception handling*: whenever the program reaches an unintended state, such as trying to divide by zero, an exception is raised and the control is transferred to a special exception handler, that recovers the program to an acceptable state.

Xept: A software instrumentation method for exception handling. This paper [107] poses the question of how to handle faults when source code is not available. As a solution to this problem, the authors suggest the Xept framework, which consists of (1) a small language to write exception handling code for functions, and (2) tools to instrument the object code using the exception handling code written in part 1.

Overall, Xept method may be considered more of a blackbox approach than a specification-based approach since only the input arguments and the return value of a function is of interest in Xept. Xept may be considered a specification-based approach only if a specification that relates the input and output arguments is provided. Only then one would be able to employ more sophisticated recovery mechanisms than the existing blackbox mechanisms (such as retry and checkpointing & recovery) presented in the paper.

8.2 Fault-Tolerance Preserving Refinements

First, in Section 8.2.1, we briefly summarize our work on stabilization preserving refinements. Then, in Section 8.2.2, we present existing methods on refinement of fault-tolerance and compare them with our stabilization preserving refinement methods.

Finally, in Section 8.2.3 we present some simulation relations developed in the literature on refinement and investigate them to see whether they are sufficient for capturing our stabilization preserving refinements.

8.2.1 Our work on stabilization preserving refinements

We have shown in [6] that refinements in general are not fault-tolerance preserving, that is, even though A is fault-tolerant, a refinement C of A may not be fault-tolerant. We are therefore led to considering special classes of refinements. In current research, we have identified two fault-tolerance preserving refinements: everywhere refinements [6] and convergence refinements [32].

Intuitively speaking, everywhere refinements demand that the implementations always satisfy the specifications from every state. Further, for effective design of fault-tolerance in distributed systems, we identify the subclass of *local everywhere refinements*: these refinements are decomposable into parts each of which must always be satisfied by some system process from *all* of its states without relying on its environment (including other processes).

Intuitively speaking, convergence refinement implies that even in the unreachable states the computations of the concrete system C track the computations of the abstract system A, although some states that appear in the computations of A may disappear in the computations of C, and hence, C preserves convergence properties (e.g., stabilization) of A.

In contrast to previous work on fault-tolerance preserving refinements, we have shown that the refinements we have identified have nice compositionality properties making them suitable for specification-based design of fault-tolerance. For example, convergence refinement enables a non-stabilizing implementation C to be made stabilizing without knowing the implementation details of C but knowing only an abstract specification A that C satisfies. More specifically, given C that is a convergence refinement of A, first stabilization of A is designed by devising an abstract wrapper W for A. Stabilization of C is then achieved by adding to C any convergence refinement of W; the refined wrapper is oblivious to the implementation details of C.

8.2.2 Previous work on fault-tolerance preserving refinements

Method by Z. Liu and M. Joseph. Liu and Joseph [76] have considered designing fault-tolerance via transformations. In their work, an abstract program A is refined to a more concrete implementation C and then based on the refined program C and the fault actions F that are introduced in the refinement process, further precautions (such as using a checkpointing & recovery protocol) are taken to render C fault-tolerant. They design the tolerance based on the concrete program, while we design our wrappers based on the abstract program.

Method by L. Lamport and S. Merz. In [70], Lamport and Merz claim that there is no need for a special technique for formal specification and verification of fault-tolerance systems, and that refinement of fault-tolerance programs could be achieved using temporal logic of actions (TLA) and a hierarchical proof method.

Towards this end, they show how a message-passing Byzantine agreement program (of [71]) can be derived from its high-level specification. (The authors, however, do not discuss how their example can be generalized into a method for designing arbitrary fault-tolerant programs.) They first present three specifications for the Byzantine agreement program: a high-level problem specification, a mid-level specification of the algorithm, and a low-level specification for message-passing model. Then they prove that each specification implements the next-higher one.

The authors claim that little ingenuity is required for proofs of refinements since a hierarchical proof strategy is adopted. However, it should be noted that a considerable amount of ingenuity is still required for coming up with the refinement programs in the first place. The authors also admit in the discussion section of the paper that their method is "not yet feasible for reasoning at the level of executable code, except in special applications or for small parts of a system."

Method by McGuire and Gouda. McGuire and Gouda [81] have also dealt with fault-tolerance preserving refinements of abstract specifications. They have developed an execution model that can be used in translating abstract network protocol specifications written in a guarded-command language into C programs using Unix sockets. Their framework solves the fault-tolerance preserving refinement problem for a guarded-command to a C program by producing a weakly-stabilizing [47] C program.

Fault-tolerance preserving atomicity refinements. Fault-tolerance preserving refinements have been studied in the context of atomicity refinement [22,85], whereas in our work we have studied them in the more general context of computation-model refinement. The fault-tolerance preserving refinements presented in [22,85] are instances of everywhere refinements; here we present a more general type of fault-tolerance preserving refinement, convergence refinement.

Semantics of fault-tolerance preserving refinements. Leal [72] has also observed that refinement tools are inadequate for preserving fault-tolerance. The focus of his work is on defining the semantics of tolerance preserving refinements of components. Whereas, in our work, we have focused on sufficient conditions for fault-tolerance preserving refinements.

8.2.3 Simulation relations and fault-tolerance preserving refinements

In [77], Lynch and Vaandrager give a unified and comprehensive presentation of simulation techniques using a simple untimed automaton model. In particular, they define refinements, forward simulations, backward simulations, and hybrid forward-backward and backward-forward simulations. Relationships between the different types of simulations, as well as soundness and completeness results, are stated and proved.

In this section we first briefly describe these simulations and then investigate whether they may be applicable for specification-based design. More specifically, we compare these simulation relations with our everywhere and convergence refinements. We show that refinement definition of Lynch and Vaandrager can be easily extended

to capture our everywhere refinements. We also show that none of the above mentioned simulations (including bisimulation) are rich enough to capture convergence refinements.

A *refinement* from an automaton C to another automaton A is a function from states of C to states of A such that

- the image of every start state of C is a start state of A,
- every step of C has a corresponding sequence of steps of A that begins and ends with the images of the respective beginning and ending states of the given step, and that has the same external action.

This definition implies that traces of C are also traces of A and is general enough to capture the notions of refinement discussed in [69, 74]. Note that if the refinement function is total and onto, then this definition is equivalent to our notion of $[C \subseteq A]$ (read C everywhere refines A), which asserts that starting from any state in C every computation of C is a computation of A. Note that we do not always require the refinement function to be onto: if we use everywhere refinements only for preservation of fault-tolerance, then a total refinement function will suffice. However, if we want specification-based fault-tolerance (i.e., if we need to design and abstract tolerance wrapper for the specification and map it to the concrete level to achieve fault-tolerance of the concrete system), the refinement function should be total and onto.

Forward and backward simulations generalize refinements to allow a set of states of A to correspond to a single state of C.

A *forward simulation* from C to A is a relation over states of C to states of A such that

- every start state of C has some image that is a start state of A,

- every step of C and every state of A corresponding to the beginning state of the step yield a corresponding sequence of steps of A (that has the same external action as the given step of C) ending with an image of the ending state of the given step.

A *backward simulation* from C to A is a total relation over states of C to states of A such that

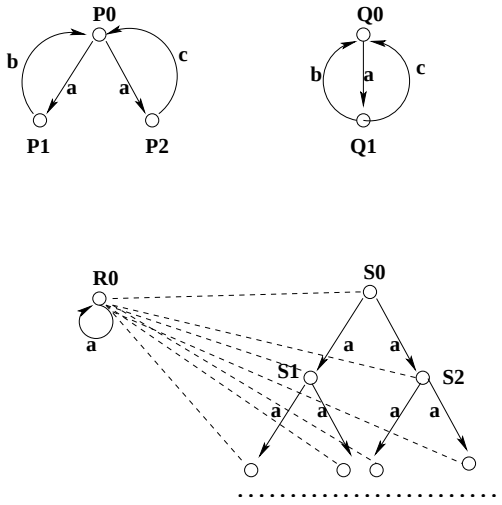
- all images of every start state of C are start states of A,
- every step of C and every state of A corresponding to the ending state of the step yield a corresponding sequence of steps of A (that has the same external action as the given step of C) beginning with an image of the beginning state of the given step.

Soundness theorems for trace inclusion preorders are proved for refinements, forward simulations, and backwards simulations. That is, if there is a refinement or a forward or backward simulation from C to A, then all traces of C are also traces of A. However, only partial completeness results can be proven for refinements, forward simulations, and backward simulations.

Forward-backward and backward-forward simulations are essentially compositions of one forward and one backward simulation in the two possible orders. Forward-backward simulations give a complete proof method for trace inclusion preorders. That is, if every trace of C is a trace of A then there exists an intermediate automaton B with a forward simulation from C to B and a backward simulation from B to A.

Bisimulations combine forward simulations in two directions (i.e., from the concrete to the abstract and from the abstract to the concrete) and are employed for proving equivalence of two systems.

Example. Below, P and Q are not bisimilar since the state $Q1$ can make both a b-transition and a c-transition and therefore is not bisimilar to neither $P1$ nor $P2$. Whereas, R and S are bisimilar and the bisimulation relation is shown by the dashed lines.



(End of example).

Summary. None of the above mentioned simulation mappings is enough for expressing a convergence refinement. These mappings always assert that for every step of the concrete, the corresponding step in the abstract should have the same external action as the concrete step. The abstract may have invisible (internal) actions that are dropped in the concrete, but visible external actions of the abstract cannot be dropped by the concrete. Whereas in convergence refinement we allow the non-initialized computations of the concrete to drop visible external actions of the corresponding abstract computations. Note that it is not possible to rename the actions that the concrete drops as invisible actions at the abstract level since the concrete does preserve those actions of the abstract in the initialized computations.

Furthermore, while an uninitialized computation $c1$ of the concrete drops the visible actions of the corresponding abstract computation a there might be another uninitialized computation $c2$ of the concrete that also corresponds to a and that does not drop any visible actions of a. We give such an example in Section 4.2 of [32].

8.3 Scalability Through Composition

Building a large-scale self-healing system via composition of self-healing components is one way of dealing with the scalability problem of the design effort of self-healing since a compositional approach reduces the global design and reasoning of self-healing to local activities at the component level.

In [73], a compositional framework for constructing self-healing (self-stabilizing) systems is proposed. The framework explicitly identifies for each component which other components it can corrupt (corruption relation). Additionally, the correction of one component often depends on the prior correction of one or more other components, constraining the order in which correction can take place (correction relation). By including both correction and corruption relations, the framework subsumes and extends other compositional approaches allowing design of fault-containing self-healing solutions. A global reset is potentially avoided and fault-containment is enabled when possible by using the correction and corruption relations to check and block certain components to prevent formation of fault-contamination cycles.

Depending on what is actually known about the corruption and correction relations, the framework offers several ways to coordinate system correction. In cases where the correction and corruption relations are in reverse directions, persistent corruption cycles may be formed: even though all the components are individiually

stabilizing, the system may fail to stabilize due to the continuous introduction of corruptions to the system via these corruption cycles. By employing blocking coordinators, the framework breaks these malicious cycles. In cases where both correction and corruption relations are in the same direction, no cycle forms and there is no need for blocking.

In this chapter, while developing lightweight and local refinements for designing specification-based self-healing to tracking, we restricted our work to the systems where both the correction and corruption relations are to the same direction in both the abstract and the concrete levels. This way we did not have to deal with addition of extra coordinators and blocking at the concrete system level. Development of lightweight and local refinements to a more general class of refinements where correction and corruption relations might vary at the abstract- and concrete-level systems is worth investigating.

Scalable design of stabilization through composition idea has been around in restricted forms before [73]. One example is the stabilization by composition of layers approach [36]. In the traditional stabilization by layers approach lower-level components are oblivious to the existence of higher-level components, and higher-level components can read (but not write) the state of a lower- level component. Components can corrupt each other, but only in a predetermined controlled way since lower-level components cannot be affected by the state of higher-level ones. Also, the order in which correction must take place is the same direction, the correction of higher-levels depend on that of lower-levels. Since both relations are to the same direction, there is no need for blocking.

CHAPTER 9

CONCLUDING REMARKS

We first summarize the contributions in this dissertation in Section 9.1. In Section 9.2, we outline possible extensions to this work.

9.1 Contributions

This dissertation addressed two orthogonal scalability problems in building fault-tolerant sensor network services: (1) the scalability of cost-overhead of fault-tolerance with respect to network size, and (2) the scalability of the design effort for fault-tolerance with respect to software size. These two research directions are complementary, and together enable a *scalable design of local self-healing for large-scale sensor network services*.

For addressing the first problem, we proposed light-weight fault-containment techniques for self-healing. By confining the contamination of faults within a small area, these techniques achieve fault-local self-healing within work and time proportional to the size of the perturbation, as opposed to the size of the network.

We illustrated our hierarchy-based fault-containment technique in the context of hierarchical tracking of mobile objects in sensor networks and achieved seamless tracking of continuously moving objects and fault-local self-healing. We illustrated

our stretch-factor based fault-containment technique in the context of clustering and achieved containment of changes and failures hitting a cluster locally within the immediate neighborhood of that cluster.

For addressing the second problem, we proposed a specification-based design of self-healing, which exploits only the specification of the system. Specification-based fault-tolerance avoids the drawbacks of implementation-specific and blackbox approaches, and allows the design of scalable, reusable, and low-cost fault-tolerance.

We established the foundations for specification-based design of self-healing by identifying two special classes of refinements (everywhere refinements and convergence refinements) that enable specification-based design of fault-tolerance. We also showed that, under suitable conditions, ordinary refinements (for which a lot of tool/compiler support exists) can be employed for specification-based design of self-healing. We illustrated the design of specification-based self-healing to our hierarchical tracking service in order to achieve scalability of design effort of self-healing with respect to the real world implementations of this tracking service.

Currently, we are focusing on applying our scalable self-healing techniques in the context of our DARPA-NEST funded "A Line in the Sand" (LITS) project. Our research group has already deployed LITS over a 100-node sensor network in a large terrain and achieved detection, classification, and tracking of various types of intruders (such as persons and cars) as they moved through the network. We are now working on scaling LITS to run over 10,000 nodes and to watch over a larger terrain. To this end we are utilizing our specification-based design method and light-weight fault-containment techniques for addressing the scalability issues in LITS.

9.2 Future Directions

Our work on fault-local self-healing and specification-based design of fault-tolerance have opened up several new directions for future research. We briefly mention some of these below.

9.2.1 On-the-fly addition of specification-based fault-tolerance

The specification-based approach enables *a posteriori* and dynamic addition of scalable fault-tolerance to unanticipated faults, and hence is essential for sensor network services that are subject to adverse environmental conditions, where unanticipated faults are the norm rather than the exception. Our research group has built a framework, DRSS, for dynamic composition of fault-tolerance wrappers; we are working on adapting this framework for enabling on-the-fly addition of specification-based fault-tolerance wrappers in LITS. For example, upon observing that message loss is becoming frequent and causing problems in our LITS service, using such a framework, we will be able to add ACK/NACK-based reliable delivery wrappers to heal our service without interrupting its availability.

9.2.2 Tool-set for specification-based fault-tolerance

Research on model-checking has provided several techniques for deducing an abstract system from a given concrete system. In addition, our work on specification-based design of fault-tolerance has provided refinement techniques that ensure that after synthesizing a fault-tolerance component for the abstract system, a refinement of this fault-tolerance component (using the abstraction technique in the reverse direction) will add fault-tolerance to the concrete system. Leveraging these results, we are

working on developing a tool-set that will enable synthesis of scalable fault-tolerance for a rich class of programs. The toolkit will accept a concrete fault-intolerant program as input, and will produce a corresponding abstract program and help the user to design fault-tolerance to this abstract program. Later, the toolkit will refine this abstract fault-tolerance wrapper to achieve fault-tolerance to the concrete program.

9.2.3 Syntax-driven fault-tolerance preserving compilers

Our notion of fault-tolerance preserving refinement depends on the semantics of the input. However, we conjecture that it is possible to identify certain syntactic constructs that imply fault-tolerance of a program. Some examples of these constructs are self-cleaning data structures (e.g., sets) and soft-state variables that are assigned fresh values periodically. We plan to develop tools that exploit such syntactic constructs in order to check whether a given program is fault-tolerant or not, and to add fault-tolerance to a rich class of programs.

BIBLIOGRAPHY

[1] Y. Afek and S. Dolev. Local stabilizer. *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 287, 1997.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 2002.

[3] I.F. Akyildiz, J. McNair, J.S.M. Ho, H. Uzunalioglu, and W. Wang. Mobility management in next-generation wireless systems. *Proceedings of the IEEE*, 87:1347–1384, 1999.

[4] A. Amis, R. Prakash, T. Vuong, and D.Huynh. Max-min d-cluster formation in wireless networks. *In Proceedings of IEEE INFOCOM*, 1999.

[5] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.

[6] A. Arora, M. Demirbas, and S. S. Kulkarni. Graybox stabilization. *Proceedings of the International Conference on Dependable Systems and Networks (ICDSN)*, pages 389–398, July 2001.

[7] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y-R. Choi, T. Herman, S. S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *To appear in Computer Networks (Elsevier)*, 2004.

[8] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[9] A. Arora, R. Jagannathan, and Y.-M. Wang. Model-based fault detection in powerline networking. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[10] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[11] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998. An extended version of this paper has been invited to *IEEE Transactions on Computers*.

[12] A. Arora and H. Zhang. LSRP: Local stabilization in shortest path routing. In *IEEE-IFIP DSN*, pages 139–148, June 2003.

[13] A. Avizienis. The methodology of n-version programming, 1995.

[14] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilizing by local checking and global reset. *WDAG94 Distributed Algorithms 8th International Workshop Proceedings, Springer-Verlag LNCS:857*, pages 326–339, 1994.

[15] B. Awerbuch and D. Peleg. Sparse partitions (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 503–513, 1990.

[16] B. Awerbuch and D. Peleg. Online tracking of mobile user. *Journal of the Association for Computing Machinery*, 42:1021–1058, 1995.

[17] Y. Azar, S. Kutten, and B. Patt-Shamir. Distributed error confinement. In *ACM PODC*, pages 33–42, 2003.

[18] S. Bagchi, B. Srinivasan, Z. Kalbarczyk, and R. K. Iyer. Hierarchical error detection in a software implemented fault tolerance (sift) environment. *IEEE Transactions on Knowledge and Data Engineering*, 12(2), 2000.

[19] S. Banerjee and S. Khuller. A clustering scheme for hierarchical control in multi-hop wireless networks. *IEEE INFOCOM*, 2001.

[20] A. Bar-Noy and I. Kessler. Tracking mobile users in wireless communication networks. In *INFOCOM*, pages 1232–1239, 1993.

[21] J. Beal. A robust amorphous hierarchy from persistent nodes. *AI Memo 2003-011, MIT*, 2003.

[22] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *International Symposium on Distributed Computing*, pages 223–237, 2000.

[23] Y. Bejerano and I. Cidon. An efficient mobility management strategy for personal communication systems. *MOBICOM*, pages 215–222, 1998.

[24] Chipcon. Cc1000 radio datasheet. `www.chipcon.com/files/CC1000_Data_Sheet_2_2.pdf`.

[25] Y. Choi, M. Gouda, M. C. Kim, and A. Arora. The mote connectivity protocol. *Proceedings of the International Conference on Computer Communication and Net- works (ICCCN-03)*, 2003.

[26] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[27] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000, 2001.

[28] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[29] R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *Static Analysis Symposium*, pages 51–63, 1995.

[30] A. Cournier, A. K. Datta, F. Petit, and V. Villain. Enabling snap-stabilization. *International Conference on Distributed Computing Systems*, 2003.

[31] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19:253–291, 1997.

[32] M. Demirbas and A. Arora. Convergence refinement. *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, July 2002. Best paper(1st/335).

[33] M. Demirbas, A. Arora, and M. Gouda. A pursuer-evader game for sensor networks. *Proceedings of the Sixth Symposium on Self-Stabilizing Systems(SSS'03)*, pages 1–16, June 2003.

[34] M. Demirbas, A. Arora, and V. Mittal. FLOC: A fast local clustering service for wireless sensor networks. *Workshop on Dependability Issues in Wireless Ad Hoc Networks and Sensor Networks (DIWANS/DSN 2004)*, June 2004.

[35] M. Demirbas, A. Arora, T. Nolte, and N. Lynch. STALK: A self-stabilizing hierarchical tracking service for sensor networks. Technical Report OSU-CISRC-4/03-TR19, The Ohio State University, April 2003.

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

[37] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.

[38] S. Dolev, D. Pradhan, and J. Welch. Modified tree structure for location management in mobile environments. In *INFOCOM (2)*, pages 530–537, 1995.

[39] M. Flatebo, A. K. Datta, and S. Ghosh. *Readings in Distributed Computer Systems*, chapter 2: Self-stabilization in distributed systems. IEEE Computer Society Press, 1994.

[40] S. Garland, J. V. Guttag, and J.J. Horning. An overview of larch. *Functional Programming, Concurrency, Simulation and Automated Reasoning*, 1993, OPTkey = , OPTvolume = , OPTnumber = , OPTpages = , OPTmonth = , OPTnote = , OPTannote = .

[41] S. J. Garland and N. A. Lynch. Using i/o automata for developing distributed systems. *Foundations of Component-Based Systems*, pages 285–312, 2000.

[42] F. C. Gärtner and H. Pagnia. Time-efficient self-stabilizing algorithms through hierarchical structures. In *Proceedings of the 6th Symposium on Self-Stabilizing Systems*, Lecture Notes in Computer Science, San Francisco, 2003. Springer Verlag.

[43] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to network embedded systems. Submitted to the ACM SIGPLAN(PLDI), June 2003.

[44] S. Ghosh and A. Gupta. An exercise in fault-containment: leader election on rings. *Information Processing Letters*, 1996.

[45] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *ACM PODC*, pages 45–54, 1996.

[46] M. G. Gouda. The triumph and tribulation of system stabilization. *Invited Lecture, Proceedings of 9th International Workshop on Distributed Algorithms, Springer-Verlag*, 972:1–18, November 1995.

[47] M. G. Gouda. The theory of weak stabilization. *Workshop on Self-Stabilization*, pages 114–123, 2001.

[48] L. J. Guibas. Sensing, tracking, and reasoning with relations. *IEEE Signal Processing Magazine*, March 2002.

[49] J. Hatcliff, M. B. Dwyer, C. S. Pasareanu, and Robby. Foundations of the bandera abstraction tools. pages 172–203, 2002.

[50] W. B. Heinzelman, A. P. Chandrakasan, and H. Balakrishnan. Application specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Networking*, 2002.

[51] M.P. Herlihy and S. Tirthapura. Self-stabilizing distributed queueing. In *Proceedings of 15th International Symposium on Distributed Computing*, pages 209–219, oct 2001.

[52] T. Herman. Self-stabilization bibliography: Access guide. Chicago Journal of Theoretical Computer Science, Working Paper WP-1, initiated November 1996.

[53] T. Herman. Super-stabilizing mutual exclusion. Technical Report TR 97-04, University of Iowa, 1997.

[54] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS*, pages 93–104, 2000.

[55] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS*, pages 93–104, 2000.

[56] Y. H. Hu, D. Li, K. Wong, and A. Sayeed. Detection, classification and tracking of targets in distributed sensor networks. *IEEE Signal Processing Magazine*, 19(2), March 2002.

[57] R. Beckwith J. Burrell, T. Brooke. Vineyard computing: Sensor networks in agricultural production. *IEEE Pervasive computing*, 3:38–45, 2003.

[58] Java technology. java.sun.com.

[59] M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states. *ACM Symposium on Principles of Distributed Computing*, 1996.

[60] S. Katz and K. Perry. Self-stabilizing extensions for message passing systems. *Distributed Computing*, 7:17–26, 1993.

[61] D. K. Kaynar, A. Chefter, L. Dean, S. Garland, N. Lynch, T. Ne Win, and A. Ramirez. The ioa simulator. Technical Report 843, MIT Laboratory for Computer Science, 2002.

[62] V. A. Kottapalli, A. S. Kiremidjian, J. P. Lynch, E. Carryer, K. H. Law T. W. Kenny, and Y. Lei. Two-tiered wireless sensor network architecture for structural health monitoring. *SPIE 10th Annual International Symposium on Smart Structures and Materials*, 2003.

[63] S. Kulkarni, A. Arora, and A. Chippada. Polynomial synthesis of byzantine fault-tolerance. *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[64] S. S. Kulkarni and A. Arora. Compositional design of multitolerant repetitive Byzantine agreement (preliminary version). *Third Workshop on Self-Stabilizing Systems (WSS 97), University of California, Santa Barbara*, 1997.

[65] S. S. Kulkarni and A. Arora. Multitolerant barrier synchronization. *Information Processing Letters*, 64(1):29–36, October 1997.

[66] S. S. Kulkarni and A. Arora. Multitolerance in distributed reset. *Chicago Journal of Theoretical Computer Science, Special Issue on Self-Stabilization*, 1997, to appear.

[67] S. S. Kulkarni and A. Arora. Once-and-forall management protocol (OFMP). *Proceedings of the Fifth International Conference on Network Protocols*, October 1997.

[68] S. Kutten and D. Peleg. Fault-local distributed mending. In *ACM PODC*, pages 20–27, 1995.

[69] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[70] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. *Third Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863*, pages 41–76, 1994.

[71] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[72] W. Leal. *A Foundation for Fault Tolerant Components*. PhD thesis, The Ohio State University, 2001.

[73] W. Leal and A. Arora. Scalable self-stabilization via composition and refinement. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2004.

[74] B. Liskov and J. Guttag. *Abstraction and specification in program development*. MIT Press/McGraw-Hill Book Co., 1997.

[75] J. Liu, P. Cheung, F. Zhao, and L. J. Guibas. A dual-space approach to tracking and sensor management in wireless sensor networks. *MOBICOM*, pages 131–139, 2002.

[76] Z. Liu and M. Joseph. Transformations of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

[77] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[78] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. *ACM Int. Workshop on Wireless Sensor Networks and Applications*, September 2002.

[79] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. *Proceedings of ACM Symposium on Programming Languages*, pages 141–155, 1983.

[80] P. Manolios, K. S. Namjoshi, and R. Summers. Linking theorem proving and model-checking with well-founded bisimulation. In *Computer Aided Verification*, pages 369–379, 1999.

[81] T. M. McGuire. Correct implementation of network protocols from abstract specifications. PhD Thesis in progress, http://www.cs.utexas.edu/users/mcguire/research/html/dp/.

[82] K. Mechitov, S. Sundresh, Y-M. Kwon, and G. Agha. Cooperative tracking with binary-detection sensor networks. Technical Report UIUCDCS-R-2003-2379, University of Illinois at Urbana-Champaign, 2003.

[83] V. Mittal, M. Demirbas, and A. Arora. LOCI: Local clustering in large scale wireless networks. Technical Report OSU-CISRC-2/03-TR07, The Ohio State University, February 2003.

[84] R. Nagpal and D. Coore. An algorithm for group formation in an amorphous computer. *Proceedings of the Tenth International Conference on Parallel and Distributed Systems (PDCS)*, October 1998.

[85] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *13th International Symposium on Distributed Computing (DISC)*, 1999.

[86] M. Nesterenko and A. Arora. Local tolerance to unbounded byzantine faults. In *IEEE SRDS*, pages 22–31, 2002.

[87] D. Niculescu and B. Nath. Dv based positioning in ad hoc networks. *Kluwer journal of Telecommunication Systems*, 2003.

[88] D. Peled and M. Joseph. A compositional framework for fault tolerance by specification transformation. *Theoretical Computer Science*, 128:99–125, 1994.

[89] S. M. Pike and P. A.G. Sivilotti. Dining philosophers with crash locality 1. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 22–29. IEEE, 2004.

[90] E. Pitoura and G. Samaras. Locating objects in mobile computing. *Knowledge and Data Engineering*, 13(4):571–592, 2001.

[91] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000.

[92] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering.*, pages 220–232, 1975.

[93] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, volume 35, pages 15–28, Banff, Canada, 2001.

[94] T. Saridakis. Design patterns for fault containment. *Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP'03)*, 2003.

[95] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[96] J. Shin, L. Guibas, and F. Zhao. A distributed algorithm for managing multi-target identities in wireless ad-hoc sensor networks. *IPSN*, April 2003.

[97] G. Simon, P. Volgyesi, M. Maroti, and A. Ledeczi. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. *IEEE Aerospace Conference*, March 2003.

[98] B. Sinopoli, C. Sharp, L. Schenato, S. Schaffert, and S. Sastry. Distributed control applications within sensor networks. *Proceeding of the IEEE, Special Issue on Sensor Networks and Applications*, August 2003.

[99] B. Sinopoli, C. Sharp, L. Schenato, S. Schaffert, and S. Sastry. Distributed control applications within sensor networks. *Proceeding of the IEEE, Special Issue on Sensor Networks and Applications*, August 2003.

[100] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.

[101] J. A. Tauber. *Verifiable Code Generation from Abstract I/O Automata*. PhD thesis, MIT, 2003.

[102] Crossbow Technology. Mica2. `www.xbow.com/Products/Wireless_Sensor_Networks.htm`.

[103] D. Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000.

[104] M. Tubaishat and S. Madria. Sensor networks : An overview. *IEEE Potentials*, 2003.

[105] G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.

[106] Vmware virtual infrastructure. www.vmware.com.

[107] K. P. Vo, Y. M. Wang, P. E. Chung, and Y. Huang. Xept: A software instrumentation method for exception handling. *Proc. Int. Symp. on Software Reliability Engineering (ISSRE)*, November 1997.

[108] T. Ne Win, M. D. Ernst, D. K. Kaynar S. J. Garland, and N. Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, pages 1–10, 2004.

[109] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27, 2003.

[110] J. Xie and I. F. Akyildiz. A distributed dynamic regional location management scheme for mobile ip. *IEEE INFOCOM*, pages 1069–1078, 2002.

[111] F. Zhao, J. Shin, and J. Reich. Information-driven dynamic sensor collaboration for tracking applications. *IEEE Signal Processing Magazine*, March 2002.

[112] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 1–13, 2003.