

MDMC - META-DETECTORS/METACORRECTORS: A
FRAMEWORK FOR THE RUN-TIME INSTALLATION
AND MAINTENANCE OF TOLERANCE COMPONENTS

A Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science in the
Graduate School of The Ohio State University

By

Ted Skidmore Donley,

* * * * *

The Ohio State University

2003

Master's Examination Committee:

Professor Anish K. Arora, Adviser

Professor Paul Sivilotti

Approved by

Adviser

Department of Computer
and Information Science

© Copyright by
Ted Skidmore Donley
2003

ABSTRACT

Detectors and correctors present a logical framework that can be used to add various levels of multitolerance to a given system. In general, they simplify the work of a designer by allowing her or him to reason about compositions of tolerance components based on the guarantees they provide, rather than on the implementations of the components themselves. Middleware services exist to provide programmers the appearance of local calling conventions in systems which actually may be distributed across process, machine, architectural and language boundaries. The same architecture that allows a middleware system to provide programmers this view may also be used to provide a wide variety of other, perhaps unanticipated services to clients of such a system. In this work, we will develop mechanisms to introduce detector and corrector components at run time into such a system in a client-invisible fashion, and explore the ramifications that this process may have on the system design. The main contributions of this work are as follows:

- A framework for the dynamic installation, modification, and removal of detector and corrector components in a middleware system
- Novel application of detector and corrector components to affect their own installation, verification, maintenance, replacement, and removal

- Identification of a large class of components that can be added to a running, distributed system using common middleware services through the use of interceptors in a client-invisible fashion

- Identification of a class of components that cannot be added to such a system in a client-invisible fashion, and strategies to work around this inherent limitation

MDMC - META-DETECTORS/METACORRECTORS: A FRAMEWORK FOR THE RUN-TIME INSTALLATION AND MAINTENANCE OF TOLERANCE COMPONENTS

By

Ted Skidmore Donley, M.S.

The Ohio State University, 2003

Professor Anish K. Arora, Adviser

Detectors and correctors present a logical framework that can be used to add various levels of multitolerance to a given system. In general, they simplify the work of a designer by allowing her or him to reason about compositions of tolerance components based on the guarantees they provide, rather than on the implementations of the components themselves. Middleware services exist to provide programmers the appearance of local calling conventions in systems which actually may be distributed across process, machine, architectural and language boundaries. The same architecture that allows a middleware system to provide programmers this view may also be used to provide a wide variety of other, perhaps unanticipated services to clients of such a system. In this work, we will develop mechanisms to introduce detector and corrector components at run time into such a system in a client-invisible fashion, and

explore the ramifications that this process may have on the system design. The main contributions of this work are as follows:

- A framework for the dynamic installation, modification, and removal of detector and corrector components in a middleware system
- Novel application of detector and corrector components to affect their own installation, verification, maintenance, replacement, and removal
- Identification of a large class of components that can be added to a running, distributed system using common middleware services through the use of interceptors in a client-invisible fashion
- Identification of a class of components that cannot be added to such a system in a client-invisible fashion, and strategies to work around this inherent limitation

Dedicated to Colleen, my family, and my teachers

ACKNOWLEDGMENTS

I am very grateful to Anish Arora, my advisor, who has provided me with the building blocks of theory upon which this work is based, a wealth of information about the principles of distributed systems, and most importantly, an appreciation for the process of breaking thoughts down to the most fundamental level possible.

I would like to thank Bill Leal, who has been extremely helpful helping me focus on concepts of component based design and computational reasoning.

I would like to thank Paul Sivilotti, who helped me acquire a framework of knowledge that has sparked my interest in this work, and has provided a model for examining current technologies within the framework of traditional theoretical modeling.

I would like to thank every professor at OSU who has taught me. They have all been excellent.

I would also like to thank the research associates who have provided infrastructure and ongoing feedback as I have worked through this project and thesis. In particular, Jason Hallstrom and Mariana Barca.

Very much I would like to thank my parents for sacrificing much to help me pursue my education, and for helping spark my interest in computer science.

Finally, I would like to thank my wife, Colleen Cebulla for years of love, support, and inspiration. She has made my academic pursuits more interesting and enjoyable.

VITA

January 28, 1966 Born - Cleveland, Ohio

1988 B.A., French Literature
Oberlin College, Oberlin Ohio

1991-1993 Software Engineer,
Consolidated Student Aid Service,
Hiram, Ohio

1996-1999 Technical Support Representative,
Checkfree Corporation,
Dublin, Ohio

1999 - 2000 Technical Support Team Lead,
Checkfree Corporation,
Dublin, Ohio

2000 - 2002 Software Engineer,
Checkfree Corporation,
Dublin, Ohio

2000 - present Graduate Student,
The Ohio State University,
Columbus, Ohio

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

Distributed Systems: Professor Anish Arora

TABLE OF CONTENTS

| | Page |
|--|-------------|
| Abstract | ii |
| Dedication | iv |
| Vita | vi |
| List of Tables | x |
| List of Figures | xi |
| Chapters: | |
| 1. Introduction | 1 |
| 2. Preliminaries | 5 |
| 2.1 System model | 5 |
| 2.2 Brief introduction to previous detector/corrector mechanisms and theory | 8 |
| 2.3 Predicates of detector and corrector components | 9 |
| 2.4 A few useful properties of detectors and correctors | 11 |
| 2.5 Definition (meta-detector/meta-corrector) | 12 |
| 3. Definitions, tools, & abbreviations | 13 |
| 3.0.1 Underlying system | 13 |
| 3.0.2 Underlying system component | 13 |
| 3.0.3 Tolerance component | 14 |
| 3.0.4 Protocol | 14 |
| 3.1 Instantiation related predicates | 14 |

| | | |
|-------|--|----|
| 3.1.1 | Relevance | 14 |
| 3.1.2 | Relevance list | 14 |
| 3.1.3 | Scope | 15 |
| 3.1.4 | Should exist | 15 |
| 3.2 | Performance related predicates | 15 |
| 3.2.1 | Capacity | 16 |
| 3.2.2 | Accuracy | 16 |
| 3.2.3 | Preparedness | 16 |
| 3.2.4 | Preferability | 17 |
| 3.2.5 | Expense/responsiveness | 17 |
| 3.2.6 | Utility based customizations | 17 |
| 3.2.7 | Overview of meta-detector predicates | 18 |
| 4. | Stepwise construction of a self-maintaining system | 19 |
| 4.1 | Universally applicable components | 20 |
| 4.1.1 | Detecting an unspecified static predicate in a static system | 20 |
| 4.1.2 | Tolerating additions | 23 |
| 4.1.3 | Maintaining Scope | 25 |
| 4.1.4 | Tolerating loss | 27 |
| 4.1.5 | Tolerating change - relaxing the remaining assumptions | 29 |
| 4.2 | Specific component examples | 31 |
| 4.2.1 | Ring based mutual exclusion | 31 |
| 4.2.2 | Server availability | 32 |
| 4.2.3 | Hot swapping | 34 |
| 4.2.4 | Motivated leader election | 36 |
| 5. | Overview of project goals | 41 |
| 5.1 | Human roles within the system | 42 |
| 5.2 | Project structure | 43 |
| 5.3 | Enumeration of projects, libraries and roles | 43 |
| 6. | Implementation mechanisms, findings, and implications | 45 |
| 6.1 | DRSS | 45 |
| 6.2 | Platform extensions | 46 |
| 6.2.1 | Mappings provided | 47 |
| 6.2.2 | Assertion of capabilities | 49 |
| 6.2.3 | Assertion of impossibility | 49 |
| 6.2.4 | Corollary to assertion of impossibility | 51 |
| 6.2.5 | Mitigating factors and workarounds | 51 |

| | | |
|-------|---|----|
| 6.2.6 | Overview of tolerance capabilities and client visibility | 53 |
| 6.3 | A system for creating programs, modifiable at run-time with high atomicity - programs, components, and guards | 56 |
| 6.3.1 | Programs | 56 |
| 6.3.2 | Program components | 57 |
| 6.3.3 | Actions | 57 |
| 6.3.4 | Resulting capabilities | 60 |
| 6.4 | Installing and maintaining tolerance components according to relevance and scope predicates - protocols, protocol groups, and templates | 60 |
| 6.4.1 | Templates | 61 |
| 6.4.2 | Protocol groups | 62 |
| 6.4.3 | Protocols | 62 |
| 6.5 | Case study : dynamic token ring | 63 |
| 6.5.1 | Identifying and connecting to relevant underlying system components | 63 |
| 6.5.2 | Controlling resource access | 64 |
| 7. | Conclusions and future work | 66 |
| 7.1 | Conclusions | 66 |
| 7.2 | Future work | 67 |
| | Bibliography | 69 |

LIST OF TABLES

| Table | Page |
|---|------|
| 3.1 Categories of meta-detector predicates and examples of protocols with which they may be used. | 18 |
| 6.1 Varying degrees of involvement with the underlying system and types of protocols that may implemented at that level. Client-invisibility is maintained for the first two levels. A client-proxy mapping requires changes only to the interface between the middleware and its users. Protocols which require access to the application-level stack most often require complex preprocessing and/or modification of the underlying system. | 65 |

LIST OF FIGURES

| Figure | Page |
|---|------|
| 4.1 System Overview. Every component in our system model begins with the sequential composition of a detector for relevance and for scope. Virtually all components would also include a component that verifies whether or not they are up as well. The same mechanism used to define sequential compositions is used to guide the physical construction, destruction, and maintenance of components contained therein. . . . | 40 |
| 6.1 Diagram of a client-invisible tolerance component's view of the system. Tolerance components reside within the interceptors and can view the target, each other, or the proxies that reference the targets. | 48 |
| 6.2 Diagram of a client-invisible tolerance component's view of the system with clients added. Although tolerance components can view the target, each other, and proxies, they do not necessarily know anything about the relationship between clients and proxies. The target in this diagram may be referred to by 1, 2, or 3 clients. Any or all of the clients may be the target itself. It is this asymmetry of knowledge that allows components to affect how a target is accessed, but not necessarily be able to control the behavior and/or view of clients. | 52 |
| 6.3 Diagram of the set of programs, program components, and actions within a system. A program may be attached to any interceptor. In turn, it may contain any number of components, each of which contains actions and state. Actions contain guards and statements. The guards of any of the actions may be strengthened or weakened at run-time, more actions can be added to a component, and more components to a program. Newly added program components have full access to the state of previously added components. In this way, components may be composed at run-time and the total system impact can be limited to the set of actions whose guards must be strengthened for safety purposes and the overhead of loading and running the new components. | 58 |

CHAPTER 1

INTRODUCTION

As more and more computing devices and systems are becoming interconnected, a variety of new technologies is emerging to facilitate the process of distributed computing. These advances free the programmer from worrying about many of the formatting and other concerns that emerge when co-operating software systems operate in different locations, languages, and perhaps architectures. There are several conceptually different types of problems that occur when these systems attempt to communicate. One of them is a problem of formatting and expression. Another category of problems stems primarily from a lack of a universal time and state. Many technologies and methodologies have been developed to address both of these categories issues. Middleware technologies, such as CORBA, RMI, .NET Remoting, or the DRSS Framework used in this work are geared primarily to address the issues of formatting and expression. However, they also include features to improve the robustness of communications. Notably, the DRSS Framework brings several additional characteristics into the equation, as it has built in support to apply custom-built interface mappings at run-time. Furthermore, it supports a much richer, semantic interpretation of the interfaces being used by components than most other middleware technologies. This

allows for many combinations of previously built software to be assembled in unanticipated fashions, and facilitates the introduction of a wide variety of other components into the system.

The problems of distributed computing related to a lack of universal time and state have been addressed in part with a combination of tools including various types of locks, monitors, and transactions. However, a significant number of unpredictable and transient faults that occur in distributed systems are more naturally controlled using more general components and methodologies, such as detectors and correctors, which we will be investigating in this work. Traditionally, the environments for these components have been the networking or hardware levels. As large-scale distributed systems become more common, and more novel methods for connecting existing systems together are discovered however, two new types of problems and opportunities arise. First, semantically guided tolerance components can provide necessary improvements in efficiency and reusability over their system-level counterparts. Furthermore, the needs for additional levels of tolerance can occur at unexpected times and in systems where constant availability is a critical issue. In this environment, we build mechanisms to introduce newly created tolerance components into the system at runtime, without affecting the underlying application-level components. We will refer to this ability to add a tolerance component into a system without changing the underlying application-level components as client-invisibility. In some instances, this property may be expensive and difficult, if not impossible to maintain. The fact that this may be done at runtime inherently reduces the amount of compiler support that is possible. Since the client, by definition, has no specific references to the tolerance code being added, it can be more difficult to see how the system operates as a whole.

If the actions introduced by a tolerance component actually contradict the intentions of the client code (for instance, if the client code contains a bug), then the resulting combination of client code and tolerance code can become very difficult to follow and interpret. However, the property of client-invisibility is also quite helpful. It is the property that middleware systems strive to provide for application programmers. It is extremely desirable in that it maximizes code reuse and increases the portability of solutions. It even allows some solutions to be applied to some systems for which no source code is available. The ability for the system to support the client-invisible addition of tolerance components at runtime also facilitates its ability to maintain itself. This provides greater system availability and reduces the human configuration burden. If a system is large and complicated, it is very likely that tolerance components will be applicable in only a portion of it. In such a case, it can be wasteful or even harmful to install the components throughout the system. To address unexpected tolerance concerns at runtime, we propose that two types of tolerance components be applied:

1. Tolerance components to maintain the system
2. Meta-tolerance components to maintain the tolerance components

We use the terms *meta-detectors* and *meta-correctors* to describe detectors and correctors that detect or correct information about themselves, other detectors, correctors, or compositions thereof. We describe several categories of tasks involved in the evolution and maintenance of a large, running distributed system, and demonstrate how meta-detectors can be an elegant and robust mechanism to accomplish these tasks. The remainder of this paper is organized as follows:

Chapter 2 provides a brief introduction to the theory of detectors and correctors, and defines a subcategory of these, which we name *meta-detectors*. In chapter 3 we describe a system into which meta-detectors may be introduced, terms that we will use to describe the evolving system, and types of concerns that they address. Chapter 4 describes how meta-detectors may be composed into a system in a stepwise fashion so as to facilitate and participate in the ongoing maintenance of other detector or corrector components. Chapter 5 describes the specific goals and system overview of our implementation of meta-detectors in Microsoft .NET and DRSS. In chapter 6, we describe broad classes of problems that either can, or cannot be solved within a similar environment. In addition, we describe other characteristics of meta-detectors designed in and for a run-time environment. Finally, in chapter 7 we summarize the findings of the project, as well as areas for future research.

CHAPTER 2

PRELIMINARIES

Our project will use state-based tolerance components to enhance the tolerance of application-level systems designed and built with middleware technology. We will refer to components of the application-level systems as underlying system components and to the state-based tolerance components simply as tolerance components. First, we will describe how we model the interactions of these two types of components in a dynamic system. Next, we will provide a brief introduction to the detector and corrector components that we use to add tolerance to the system. Finally, we will define the meta-detector and meta-corrector tolerance components that we use to maintain tolerance components within the system.

2.1 System model

We will consider the system to consist of user programs, the framework for allowing them to communicate, and tolerance components we will add. Although we do not anticipate that the user programs will be made of actions built from guarded commands, we will model types of behavior of the system with a few generic guarded commands. We will only specify those system actions that affect the values of the variables of our tolerance components, or are affected by the actions of our tolerance

components. In this way, we can model the introduction of a specific, reusable component into a program that may contain an arbitrarily large, but finite set of actions and still be able to reason about whether guarantees of interest are met. Users of the underlying system are assumed to have no awareness of the tolerance components. Since the system is dynamic, we need to consider how to model components that do not exist yet or that have failed. Every component is implicitly given two bits to model this.

An add action, therefore, can be modeled as follows:

$$(\neg \text{exists}.j \wedge \neg \text{failed}.j) \rightarrow \text{exists}.j$$

A component failure can be modeled as follows:

$$(\neg \text{failed}.j \wedge \text{exists}.j) \rightarrow \text{failed}.j$$

Each of these actions will be enabled only one time per component. No other actions may write to these variables. Attempts to read these variables will not return if *exists* is false or *failed* is true. Our tolerance components components have another bit associated with them which we name *shouldexist*. This bit is set by the underlying system and can neither be viewed nor altered by the tolerance components directly. However, the tolerance component may invoke one of the actions above on another tolerance component. Each tolerance component has a set of all possible underlying system components associated with it. Creating a tolerance component *cmp* and associating it with process *j* has the following effect.

$$(\neg \text{exists}.j.\text{cmp}) \rightarrow \text{exists}.j.\text{cmp};$$

However, we do not believe that it is necessarily realistic for tolerance components to create a witness for this value directly. We therefore propose that each tolerance component have two offset fields associated with it, a *relevance* field, and a *scope* field. The concatenation of these two fields uniquely identifies a tolerance component, and the association of these fields with their respective *shouldexist* bits identifies whether or not a specific tolerance component should be added, removed, or modified.

The tolerance components are able to observe but not prevent the creation of components in the underlying system. Tolerance components may view the public and private state of system components, as well as the messages they send and receive, but not necessarily the state of temporary variables. Tolerance components are not necessarily able to observe failures of underlying system components. Users of the underlying system may create new types, but not destroy them. We assume a finite, but unknown bound on message delay. Components of the underlying system capable of sending and receiving messages will be treated as separate processes for the purposes of this work. Tolerance components may be created and associated with specific underlying system components. These components share the same system clock and view the messages sent and received by the underlying system components whose behavior they are controlling. A property that such a tolerance component is able to observe about the underlying system component will be labeled as a property of the underlying system component, for example $Z.j$. Although the mechanisms for finding the value of Z are varied, we will restrict the user to construct predicates whose values are well-defined and that can be calculated in finite time. The evaluation of this predicate returns a single bit value. Our reasoning will be based on the value of this bit.

Invocations are assumed to be well-formed, and directed to the current state of the underlying system components, not the computations of which they are a part. Inquires as to the existence of a component may only be directed to a safe and well-known witness for that component. Invocations made on components that do not exist will have no effect on the system and will not return to the caller. We do not distinguish between fail-stop faults and crash faults. We will focus primarily on two classes of faults for this work, timing faults and crash faults. We choose these because repairs to these faults, even if they include transient errors, can be performed by only reading from and not writing to the underlying system, and therefore do not introduce new classes of faults directly into the underlying system. We assume that actions are scheduled with weak fairness, or that if an action is continuously enabled, it is executed infinitely often.

2.2 Brief introduction to previous detector/corrector mechanisms and theory

Detectors and correctors are formal mechanisms to provide various levels of tolerance to a system. Intuitively, detectors are formalisms that are used to provide realistic approximations of predicates that may be difficult or impossible to detect directly. They perform this by means of a witness predicate, whose value is easier to detect, and whose truth is related in some fashion to the underlying predicate whose value we really wish to know. There are many formalisms for these components, including proposals in [1], [2], [3], [4] and [5] with varying degrees of power and limitations. The common theme behind these formalisms is that we can identify a witness predicate whose value will eventually reflect that of the actual predicate in some way. The formalism that we will use for this work, defined by Arora and

Kulkarni in [6] and elaborated in other works such as [7], and [8] is defined using three predicates and a program.

2.3 Predicates of detector and corrector components

The first predicate, named U , is the predicate that must be true in order for the relationship of the other predicates to be valid. The second, X , is the actual predicate of interest that arises from the problem specification. The third, Z , is a witness predicate, whose value will be used as a surrogate for the actual predicate of interest. The program, d is the program in which the detection is desired to hold. If the component of interest is a corrector, then the program is labeled c . A program which maintains the properties of detection or correction defined below is a detector or a corrector.

Definition (detector).

Z detects X in d for U iff the following three properties hold:

Safety - $U \wedge Z \rightarrow X$

Liveness - $U \wedge X \wedge \neg Z$ leads-to Z

Stability - $U \wedge Z \{d\} Z \vee \neg X$

The $\{d\}$ notation indicates that any single step in the program has occurred. The safety property asserts that a detector is only incorrect when its witness is false. In some instances (such as a failure detector), this property may be a property to which the detector converges, rather than holds immediately and continuously. This technicality has a minimal effect on the realistic use of these components. The liveness

property simply states that the detector will not permanently fail to detect the predicate of interest if it is true continuously, and the stability property states that once true, the predicate will never become false unless the actual predicate is also false. A corrector has the same definition as a detector with a single, stronger guarantee.

Definition (corrector).

Z corrects X in c for U iff Z detects X in c for U \wedge U converges to X in c.

In short, a detector guarantees that you will at least have eventual knowledge if a certain condition becomes true and a corrector guarantees that a certain condition will eventually become true and remain true.

In many instances, it is easiest to combine simple detectors and correctors in order to accomplish the task of detecting or correcting a more complicated or difficult predicate. Two techniques for the composition of detectors and correctors are defined, parallel and sequential. As expected, a sequential composition of two detector or corrector components is one in which one detection or correction witness becomes true before the actions of the second component are enabled, and a parallel composition is one in which the actions of both components are permitted to occur at the same time. Parallel composition is denoted with the \parallel operator, and sequential composition is denoted with the $;$ operator. These have many useful properties described in [6], but for the purposes of this work we will focus primarily on the following synopses.

2.4 A few useful properties of detectors and correctors

Compositions of detectors and correctors may be chained together in order to build stronger guarantees from weaker ones. We focus primarily on sequential compositions because we have used these most often in the preliminary stages of our system construction.

*If Z detects X in d for $U \wedge$
 W detects Y in c for $X \wedge$
 $c \wedge d$ do not interfere with each other
then $Z \wedge W$ detects Y in $d;c$ for U .*

This states that, given the sequential composition of a detector followed by either a detector or corrector, the conjunction of the witnesses detects the final predicate of interest in the global scope. In general, non-interference occurs when c and d do not write to variables that the other reads, but a more formal and complete set of conditions for non-interference is described in [6].

*If Z corrects X in d for $U \wedge$
 W detects Y in c for $X \wedge$
 $c \wedge d$ do not interfere with each other
then W detects Y in $d;c$ for U .*

This is a stronger property. Given the sequential composition of a corrector followed by either a detector or corrector, the final witness detects or corrects the final

predicate of interest in the global scope accordingly. Taken together, these two theorems state that non-interfering detectors and correctors may be chained together sequentially. If both are correctors the result is a corrector in which the final witness corrects the final predicate in the universal scope. If the first one is a detector, then the conjunction of the witnesses detects the final predicate in the global scope, otherwise the final witness detects the final witness in the global scope.

We will make one other observation that follows directly from the definition of detectors and correctors. Every detector or corrector is the sequential composition of itself and a detector for its U predicate. We will use this property to guide the stepwise construction of our system.

2.5 Definition (meta-detector/meta-corrector)

In this work we will concern ourselves with a particular type of detector or corrector component, which we name a *meta-detector* or *meta-corrector*. A detector or corrector is a meta-detector or meta-corrector if it detects or corrects information pertinent to the installation or maintenance of another detector or corrector component. We will show that the application of these components to other tolerance components provides a sound and flexible mechanism to add dynamically self-regulating components to a system at run-time.

CHAPTER 3

DEFINITIONS, TOOLS, & ABBREVIATIONS

In this section, we will define a set of terms that help to clarify the structure of the system, the nature of predicates inherent to meta-detection, and some of the capabilities it can provide.

3.0.1 Underlying system

We use the label *US* to refer to the underlying system, and also as a label for a program that models underlying system actions. Actions that are assumed to be generated by the system are labeled *US1...N* where the indexes 1-N are labels for different classes of actions described above and enumerated in the program actions that model the system.

3.0.2 Underlying system component

An object in a high-level language that handles application logic and is capable of sending and receiving messages.

3.0.3 Tolerance component

Tolerance components are added by an administrator or manager to add tolerance to a system. In most cases each tolerance component will be associated with a particular underlying system component or another tolerance component.

3.0.4 Protocol

A set of components co-operating to achieve a specific goal on a specific set of components.

3.1 Instantiation related predicates

Within the system, we would like to be able to instantiate detector and corrector components according to a set of criteria, as opposed to specifically invoking instance methods on specific instances of components. This allows us to create components within a system that take responsibility for their own ongoing maintenance according to the rules specified by predicates that witness these criteria.

3.1.1 Relevance

The relevance predicate is the predicate that defines whether or not a particular component is deemed applicable in a particular problem scenario. For example, if a given type of resource “requires mutual exclusion”, or “requires replication”, these would be relevance predicates.

3.1.2 Relevance list

This is a list of all of the components deemed to be relevant in a particular problem scenario. It also used as a label for program actions that maintain the list. Since this

part of the system is under our control, each action that we will describe, (e.g., RL1, RL2, etc.) will describe a specific action in a program dealing with a relevant set of components within the system.

3.1.3 Scope

This is set of components to which a particular instance of a protocol applies. For instance, if there are several resources that require mutual exclusion, the scope of each protocol that manages access to each resource would be the resource itself, and the set of all its users.

3.1.4 Should exist

This is an imaginary single bit field that indicates whether a particular instance of a tolerance component, associated with a particular system component should exist or not. We stipulate that a tolerance designer is not able to access this bit directly. In fact, we expect this inference to be made only through witnessess of two other bits, a relevance bit and a scope bit. The pair of these may be used to uniquely identify a given tolerance component that should exist as well as to possibly determine when a given component should not exist.

3.2 Performance related predicates

Having established a set of components within a system, we expect that a wide variety of circumstances may arise when these components need to update themselves according to certain criteria in order to best fulfill the requirements of the underlying system. We contend that the following are some of the types of predicates that can be used in order to help tolerance components better fulfill these requirements.

3.2.1 Capacity

Many detectors or correctors, such as the server availability protocol proposed by Vijay Garg [2], are guaranteed to meet certain guarantees in the event that they have a certain capacity (in this case, enough servers so that they do not all fail). These values may be determined and adjusted by correctors of the component itself, and therefore ensure that the capacity requirements of the tolerance component will be met, allowing it to meet the tolerance requirements of the underlying system.

3.2.2 Accuracy

Standard protocols exist to enforce certain predicates within a system according to certain criteria. Generic predicate protocols, similar to the one proposed by Cooper and Marzullo [9] or Stoller [5] may be used to verify the accuracy of the guarantees supplied by and/or required from a given detector or corrector, either at the current moment or over the history component.

3.2.3 Preparedness

In addition to being relevant for the set of components to which it is applied, a tolerance component may also require that certain other preconditions be met before it is activated. An example would be the event in which we wish to replace one component with another. In this event, we need to make sure that any actions of the first which may negatively effect the correctness of the replacement not be enabled before the replacement is activated.

3.2.4 Preferability

Given the ability to satisfy user requirements with more than one implementation, we are faced with the possibility that one may perform the job more efficiently than the other. In a dynamic system, it is also quite possible that the preferability of one method to another may change at run-time. To the extent that we are able to detect this and safely swap one component with another, we can create a system that automatically chooses the best current approach to a given problem.

3.2.5 Expense/responsiveness

We remark that all tolerance components come a certain cost of system resources. By examining the amount of resources that it is using to detect or correct a certain condition, a tolerance component may be able to reduce the burden it places on the underlying system while still providing comparable guarantees. If a particular guard is evaluated an average of thousands of times before its value is ever true, it may very well be able to achieve its goal nearly as quickly with a much smaller system burden if it is scheduled much less often. Similarly, if a guard evaluates to true every single time that it is evaluated, that may be an indication that it is not being scheduled often enough for the program to be satisfactorily responsive to changes in the environment. Given requirements and detectors for responsiveness and/or efficiency, we may regulate the scheduling in such a fashion so as to better meet these requirements.

3.2.6 Utility based customizations

In many instances we see that a component designed for a certain purpose can be re-worked to provide a component that works well in a different or more specific situation. These typically depend on the individual orderings of components and can

be reapplied to components whose orderings have the same properties. Scott Stoller shows an example of this in [5]. We will show that it is possible to use a set of detectors and correctors to construct an ordering based on any criterion that has a sufficiently stable relation to the underlying components, and show how we might use such a construct to adapt the utility of a given component at run-time.

3.2.7 Overview of meta-detector predicates

We will next show how meta-detector components meeting many of these criteria can be used to install, maintain, and enhance other components through the stepwise construction of a partially self-maintaining system. A summary of types of tasks that might be performed by meta-detectors and clients that may collaborate with them is provided in table 3.1

| Type of Predicate | Motivating Example |
|--------------------------|---|
| Relevance | Universally applicable |
| Scope | Universally applicable |
| Capacity | Server availability |
| Preparedness | Hot-swap of mutual exclusion protocols |
| Utility | Customizable ordering for leader election protocols |

Table 3.1: Categories of meta-detector predicates and examples of protocols with which they may be used.

CHAPTER 4

STEPWISE CONSTRUCTION OF A SELF-MAINTAINING SYSTEM

We will demonstrate how meta-detectors and meta-correctors are used to install and maintain tolerance components within a running system. Our strategy will be to start by building tolerance components for a static system in which detection of the predicates we are interested in is very straightforward. Next, we will enhance our components to be able to perform their work successfully with an underlying system that allows for the addition of new components, removal of existing components, and the modification of existing components. Unless otherwise specified, we will require that each new component we introduce into the system may read, but not write the variables of any existing component within the system. This will prevent our system from interfering with existing components, and allows us to reason about each addition in isolation. In some situations, we must write to the variables of previously installed components, and therefore must argue that the system goals are met despite this interference in those cases.

4.1 Universally applicable components

Two of the types of predicates we will investigate, relevance and scope, are used in the installation and maintenance of all other tolerance components. Their purpose is define the location and initial parameterization that present a tolerance component with the set of states in which it is designed to work. A detector for relevance will be used to define the set of states in which a detector for scope is allowed to run, and a detector for scope will be used to define the set of states in which a tolerance component composed sequentially with it is allowed to run.

4.1.1 Detecting an unspecified static predicate in a static system

We will begin by defining a component for relevance. Intuitively, relevance defines the subset of a system in which a certain component is applicable. We will suppose that there is a predicate X that defines what it means for a particular component in a system to be relevant. We will also suppose that there exists a predicate Z that can successfully witness that this element satisfies the predicate X for U where U is the predicate true (We will later show that we can relax this restriction, by allowing U to be a more restrictive predicate). When necessary, we will denote the boolean value that can be obtained by evaluating these predicates at a particular underlying system component j as $X.j$, $Z.j$, or $U.j$ respectively. Finally, we will temporarily assume that X is fixed in U . (We will also later remove this restriction as well). We will name a system that meets these assumptions US . Since we are only modeling actions that affect the predicates of our detector and corrector components, and we are currently assuming X is fixed in U , our program currently has no actions.

We will state by convention $\neg(\text{exists}.j) \rightarrow \neg X.j$, where X is the relevance predicate of interest. Stated another way, we will not attempt to add tolerance components to components that do not exist (and we will also remove tolerance components associated with components that do not exist, which we will see later).

Lemma 4.1.1.1 A detector's witness predicate eventually becomes equivalent its predicate of interest in a system with no underlying actions.

Z detects X in US for $U \wedge$

Z corrects $(X \vee \neg Z \wedge Z \vee \neg X)$ in US for U .

Proof:

Our obligation is to show that safety, liveness, and stability hold, to prove that the detection holds and that U converges to $X \vee \neg Z$ to show that the correction holds. We know that safety $U \wedge Z \rightarrow X$, liveness $U \wedge X \wedge \neg Z$ *leads-to* Z and stability $U \wedge Z \{d\} Z \vee \neg X$ hold because we have directly assumed that Z is able to witness X for every component in the system. Convergence to $X \vee \neg Z$ follows directly from the definition of a detector. Convergence to $Z \vee \neg X$ is proven as follows: By the liveness property of detectors, we know that $Z \vee \neg X$ will eventually hold. By the stability property of detectors, we know that it will continue to hold since the program so far contains no actions that might falsify X .

Given this assertion, and a mechanism to install a component that evaluates Z at any component j , we can install a single-action program at every component j in the system as follows:

$RL1:: Z.j \rightarrow list.RL := list.RL \cup j$

This program possesses the following useful property: If RL1 and the system do not interfere with each other, then we have the following:

Preliminary detector for Relevance

$j \in list.L$ detects $X.j$ in $US \parallel RL$ for true.

We also have the stronger and desirable property at this point:

True converges to $(j \in list.L) \vee \neg X.j$ in $US \parallel RL$ for true.

Starting from any system state under the assumptions above, therefore, we have a detector that eventually stops making both positive and negative mistakes. We shall prefer this stronger detector when possible because it provides a stronger guarantee, and therefore a lighter proof burden on components composed sequentially with it. We can also view RL1 as a corrector.

Lemma 4.1.1.2

$(j \in list.L) \vee \neg X.j$ corrects $(j \in list.L) \vee \neg X.j$ in $US \parallel RL$ for true.

Proof: Since each component is added to the list only if X has been detected at that component and X is fixed we know that the safety property holds. Since X is closed in U , we know, by the assumption of weak fairness, that every component for which X is true

will eventually be added to the list, and that liveness holds. Since no component is ever removed from the list, we know that stability holds. Since the list eventually becomes complete and never changes once every element has been added, we know that the convergence property of correction holds.

We prefer to view this as a detector with a stronger convergence property since the component is not intended to change the underlying system, just provide information about where other tolerance components may be installed. There are many conditions under which the non-interference condition between US and RL will not hold, but we also may define a rich class of conditions where they do. To begin with, they hold if the actions of RL may read, but do not write to the variables used by the underlying system, and the underlying system neither reads nor writes the variables of RL. In short, we are superposing RL on the system, to get a new system with the same properties and also a list, which is guaranteed to eventually contain all elements matching the relevance criterion.

4.1.2 Tolerating additions

A middleware service allows for the addition of underlying components. We will now add an action that adds a new underlying system component into our system with the minor restriction that it not falsify the value of our predicate of interest at any other component.

$$US2 :: (\neg exists.j \wedge \neg (\exists k \mid X.k \wedge exists.j) \rightarrow \neg X.k) \rightarrow exists.j$$

For ease of exposition, we will paraphrase the above action and other addition requests in the remainder of this work. As it is, the former corrector no longer maintains closure in the system containing the action US2. However, since US2 is part of the middleware service over which we have some control while maintaining client-invisibility, we can modify the action US2 as follows:

$$US2a :: (\text{request to add } j) \rightarrow \text{AddCheckRL}; \text{ add } j \text{ to system}$$

Where *AddCheckRL* is a program whose actions are performed simultaneously at the moment a new component is added to the system and that initially contains the following action:

Lemma 4.1.2.1 $j \in \text{list.RL}$ detects $X.j$ in $US \parallel \text{RL} \parallel \text{AddCheckRL}$

$$\text{AddCheckRL1} :: (US = US \cup j; \forall k \mid Z.k \rightarrow \text{add } k \text{ to list.RL})$$

Proof:

Since we are assuming that X is closed in U , safety follows from the fact a component k is only added to the list if $Z.k$ is true. Liveness follows from the fact that every component in the system is verified every time any addition which may truthify X occurs. Stability follows from the fact that we never remove any item in the list.

Although in the worst case, this check is extremely expensive (potentially the entire system), the most common case involves only a test of the component being added and perhaps its immediate neighborhood, making the test very efficient. The system no longer allows the add request of the underlying system to violate the

closure of the witness predicate of our first detector, and our assumptions now allow the system to expand without affecting the correctness of the relevance list RL.

4.1.3 Maintaining Scope

We have defined a set of actions that allow us to maintain a list for which a particular type of tolerance component is applicable. As we introduce tolerance components into the system, we will see that multiple components may be introduced to affect the region of relevance, and that the set of underlying system components maintained by each of these, or scope, may also change as the system expands or contracts. We describe a general mechanism to assign each system component to a unique scope. We will assume that we can identify a predicate to witness whether a relevant underlying component is also in the scope of a given tolerance component. We label this predicate $Z.cmp$. We label the actual predicate, that an element belongs in the list of cmp (the list of underlying components controlled by the given tolerance component), as $X.cmp$. We will also define an operation that creates a new component cmp , such that $Z.cmp$ is true when this component's list contains the single element j , and name this operation $createCmp$. We name the list of underlying system components controlled by a single tolerance component or protocol $list.cmp$ and list of tolerance components to which they belong $cmpList.RL$. We sequentially compose actions with the program $AddCheckRL$ as follows:

$$ScopeCheckRL1 :: (\forall cmp \mid cmp \in cmpList.RL \ Z.cmp \rightarrow list.cmp = list.cmp \cup j)$$

$$ScopeCheckRL2 :: (if \neg \exists cmpList.RL \mid Z.cmp) \rightarrow createCmp; list.cmp = list.cmp$$

$$\cup j; cmpList.RL = cmpList.RL \cup cmp$$

Lemma 4.1.3.1

$j \in \text{list}.cmp$ detects $X.cmp$ in ScopeCheckRL for X

Proof: Safety is preserved since no action adds a component unless its witness is true. Liveness is guaranteed since every system component is associated with some tolerance component for scope and we assign each system component to a unique scope. Stability is guaranteed because no item is ever removed from a list.

Lemma 4.1.2.2

$j \in \text{list}.cmp$ detects $X.cmp$ in $US \parallel RL \parallel \text{AddCheckRL} ; \text{ScopeCheckRL}$ for U and $true$ converges to $(j \in \text{list}.cmp \vee \neg X.cmp)$ in $US \parallel RL \parallel \text{AddCheckRL} ; \text{ScopeCheckRL}$ for U

Proof:

At any time in the system, the potential size of any list is finite, since each component in each list is associated at most once with a finite set of underlying system components that have been created. We first note that we can represent membership in a list as a single bit boolean value for the purposes of compositional reasoning. Since this detector is the sequential composition of the relevance detector for X and the scope detector for $X.cmp$ we know that each component's list eventually contains every component that is continuously relevant and in scope, by theorem 3.3 in [6]. Since no action of the US or the added components (that has been permitted so far) falsifies $X.cmp$, we know that eventually each list will contain all relevant and in scope components. We note also that the physical construction of lists is not necessary, but may be helpful in some sequential compositions where the second component requires global knowledge of the set of components identified by the witnesses of the the first.

4.1.4 Tolerating loss

Now that we have a system that adapts to additions of components that are relevant and in scope, we wish to enhance it so that it adapts to losses of these components as well. We will imagine that every component contains an imaginary bit that indicates whether or not it has failed. We will use a detector for this failed bit to define a component that can be used to further restrict the set of states in which a tolerance component may be allowed to run.

Failure detector

We have yet to consider what happens when components fail, a likely occurrence in any large-scale system. Our default implementation will be similar to the one found in [1] and [2] among others, but will be tailored to help maintain the relevance and scope lists developed thus far. In this case, the witness function is not application-dependent, only the elements to which it applies are. The actions of the detector are as follows:

On each component being checked:

$FD1 :: (intervalElapsed) \rightarrow \text{send alive message to server}$

At the server:

$FD2 :: (now - lastMessage.j > timeout.j \wedge j \text{ is in } list.RL) \rightarrow \text{remove } j \text{ from } list.RL, list.cmp; \text{add } j \text{ to } list.FD;$

$FD3 :: (\text{hearMessage from } j \wedge j \text{ is in } list.FD) \rightarrow \text{timeout} = \max(\text{timeout}, \text{now} - \text{lastMessage.j} + 1); \text{remove } j \text{ from } list.FD; \text{add } j \text{ to } list.RL, list.cmp;$

This can be viewed as a failure detector, by stating:

Lemma 4.1.4.1 *j is an element of $list.FD$ detects that j has failed.*

This satisfies all but the safety property $Z \rightarrow X$ (this detector initially suspects everything). Under the assumption that the time to send a message and the clock drifts are bounded, but not necessarily known, eventually this detector will satisfy the safety property of a detector as well.

Proof: Safety follows from the assumption that there is a bound on the message delay. At some point the detector will increase its timeout until its value exceeds that delay, and therefore will hear every message from every process that has not failed before that delay has occurred. All such components will be removed from the detector's list of suspects and will not be added to the list of suspects in the future. Therefore, there exists a time after which all components which are suspected have actually failed. Liveness follows from the fact that a failed process sends no messages, and therefore will eventually be added to the suspect list. Stability follows from the fact that once a failed component has been suspected, it will never be removed from the list.

It can also be viewed as an up detector by stating:

j is not an element of $list.FD$ detects that j is up;

This also violates the pure safety property of a detector. In fact, it will sometimes report false positive information about the status of a failed process as long as processes continue to fail. However, we do still know that the liveness and stability properties eventually hold for this detector under the assumption of partial synchrony. Furthermore, we state that this is a *nonmasking* detector, since, if processes stop failing, it eventually meets the full specification of a detector [6]. To ensure that new

additions are also subject to failure detection, we add the following action to the *ScopeCheckRL* program.

$$\text{ScopeCheckRL3} :: (\text{if } \exists \text{ cmp in cmpList.RL} \mid \text{there does not exist FD.cmp}) \rightarrow \text{createFD};$$

4.1.5 Tolerating change - relaxing the remaining assumptions

By modifying two of the actions to FD, we can maintain a system that eventually maintains correct lists under partial synchrony once faults stop occurring (we model a change in relevance for a component as a fault), even if the relevance witness Z depends upon variables that are not constant under the underlying system. We will do so by strengthening the guards on some of the actions of our failure detector as follows:

On each component being checked:

$$\text{FD1a} :: (\text{intervalElapsed} \wedge Z.j) \rightarrow \text{send alive message to server}$$

At the server:

$$\text{FD3a} :: (\text{hearMessage from } j \wedge j \text{ is in list.FD wedge } Z.j) \rightarrow \text{timeout} = \max(\text{timeout}, \text{now} - \text{lastMessage}.j + 1); \text{remove } j \text{ from list.FD; add } j \text{ to list.RL, list.cmp};$$

Lemma 4.1.5.1

$j \in \text{list.cmp}$ detects $X.cmp$ in $US \parallel RL \parallel \text{AddCheckRL} ; \text{ScopeCheckRL} ; \text{FD}$ for U

Proof: Our obligation is to show that once faults stop occurring, after a finite amount of time, the safety, liveness, and stability properties will hold. To verify safety, we first observe that if any component is in *list.cmp* that is not up, relevant, and in scope, it will eventually be removed by FD2. Any component is added only if it is up, relevant and in scope. Liveness and stability follow from the fact that any component that was removed from *list.cmp* but eventually and continuously remains, up, relevant, and in scope will eventually be added back into *list.cmp* by FD3a, and will no longer be removed by any other action.

Lemma 4.1.5.2

$j \in list.cmp$ corrects $SPEC.cmp$ in $US \parallel RL \parallel AddCheckRL ; ScopeCheckRL ; FD ; cmp$ for U

Proof: The component *cmp* is specifically defined to be a non-masking tolerant corrector for *SPEC.cmp* in the scope specified by *list.cmp*. Since *cmp* is non-masking tolerant, it is guaranteed to converge to its specification once faults stop occurring. Since the above detector is non-masking tolerant, it is designed to converge to its spec once the faults stop occurring. Once this occurs, *cmp* will no longer receive any input outside of its input specification and will converge to its specification.

In short, we have a set of detectors, in the form of a set of lists, which will guide the installation and maintenance of tolerance components according to criteria specified by the users. We have also shown that list maintenance provides a simple and intuitive physical tool for the sequential composition of detectors, and that the reasoning tools developed for the composition of detectors may be applied to this same maintenance. For the case where the only change in the set of elements matching the criteria for list membership is the addition of a new one, we have shown that these lists possess

the very strong property of converging to precisely correct membership. For other cases, we have shown that once the changes stop, the system converges to precisely correct membership.

4.2 Specific component examples

Having established a self-maintaining, predicate-based mechanism to introduce tolerance components into a running distributed system, we will look at some examples that have been generated or may be generated showing how the model works, and additional techniques that may be used for detector and corrector components to affect their own runtime evolution.

4.2.1 Ring based mutual exclusion

We will begin with an alternating bit protocol, shown by [10] to be usable as a sliding window or for mutual exclusion, and to extend to a k-state token ring protocol by simply increasing the maximum size of the counter incremented at each turn. We will consider the case that we wish to use the component for mutual exclusion. In such a case, the predicate of interest, or X , is that the underlying system component possesses an attribute that it must be accessed in a mutually exclusive fashion. The witness predicate, or Z will reflect whatever properties of a component can most reliably lead one to the conclusion that the predicate of interest holds. For this case, we will assume that this may be determined from the type of the underlying component, and in our example will be $(type.j = typeof(Resource) \vee (adj.j.k \wedge type.k = typeof(Resource)))$. To build the list of relevant components, therefore, we need only supply the system with the type of component being added (In this case mutual exclusion), and the witness predicate (shown above). To allow each component to

maintain its scope we specify the following scope criterion as $Z.cmp$: (*resource used by inbound = currentResource*). This causes the system to create a new protocol for every new resource, and to add new users to every protocol which has already been created for that resource. In addition, the ring component must specify how to add and remove elements, as well the implementation to enforce mutual exclusion. We will discuss the details of how the protocol does this in the project discussion section of this work.

4.2.2 Server availability

Another need that might arise in a large, distributed system is the need to replicate servers to be certain that at least one is available, even in the event that servers may crash. In this case the predicate of interest would be that the underlying system component possesses the attribute of requiring replication. The witness predicate could be ($type.j = typeof(constrainedserver) \wedge numUsers.j > maxUsers$) and the type of component being added would be a replicator. The scope predicate could be (*server used by inbound = originalServer*). The protocol would be responsible for creating replicas of the original server in this instance, and applying the solution to each of many replicas created. In this case, the scope of the protocol would be the original server created by the application designers, its users, and the replicas created by the protocol. This example also illustrates another way in which the scope of a component may be adjusted at runtime. The protocol is responsible to create replicas of a given server, but does not know how many replicas may be necessary at a given time. With experimentation, a system manager can find a number that seems to work well. However, a detector and corrector may serve better for this purpose as long as the

number of servers that may fail in any period is finite, and for some finite number k and any positive number n , the system can create $k + n$ servers faster than $k + n$ servers may fail. Suppose that a replicator is designed to be certain that at least one server is always available and already contains an action as follows:

$RP1 :: (numServers \leq minThreshold) \rightarrow while (numServers \leq maxThreshold)$
createServer elihw

We can add the following action:

$RP2 :: (numServers \leq minThreshold/2) \rightarrow minThreshold += 1; maxThreshold =$
 $minThreshold * 3/2;$

Lemma 4.2.1 In the above program:

(numServers > minThreshold/2) nonmasking corrects there are sufficient servers for U in RP.

Proof:

Our obligation is to show that the program converges to a state that contains more than 0 servers. We do this by showing it reaches a state where more than $minThreshold/2$ are maintained. Whenever this is not true, the system increases both the min threshold, and the max threshold. The protocol will converge when it can create $minThreshold/2$ servers before that many can fail. Assuming that the rate at which servers can fail is finite and in the long run lower than the rate at which servers can be created (which will be the U predicate), the program will eventually reach a point where the action RP2 is never

executed, and the number of servers necessary for the server availability protocol to be successful will be maintained.

4.2.3 Hot swapping

The need may arise, for reasons of necessity or efficiency, to swap one component with another that performs the same function by means of a different implementation. In one scenario a system manager might temporarily block one portion of a system, remove the old component, install the new one, and allow the system to continue. However, in many cases it is not necessary to do this in order to accomplish the goal of having one component be replaced by another. In the case of token protocols, for example, we may introduce predicates *ready-to-remove*, and *ready-to-add*, at each subcomponent within the replacee and replacer accordingly. We assume neither protocol violates the safety predicate (that only one token may exist in the system at a given time) and that the new protocol also respects the liveness predicate (that every component eventually receives the token). We also assume that exactly one token exists in the old protocol. In addition, the components will maintain counts of the number of subcomponents within their protocols, creating a set of actions as follows:

$$\begin{aligned} SWP0 :: & (receivePseudoToken.j \text{ from } k \wedge readyToAddCount.swp.j = 0) \rightarrow createSwp.j; \\ & readyToAddCount.swp.j = readyToAddCount.swp.k + 1; \\ & readyToRemoveCount.swp.j = readyToRemoveCount.swp.k + 1; \end{aligned}$$

$$\begin{aligned} SWP1 :: & (receivePseudoToken.j \text{ from } k \wedge readyToAddCount.swp.k > readyToAddCount.swp.j > 0) \rightarrow readyToAddCount.swp.j = readyToAddCount.swp.k; \\ & readyToRemoveCount.swp.j = readyToRemoveCount.swp.k; \end{aligned}$$

$SWP2 :: (readyToAddCount.swp.j = numCount.cmp.j) \rightarrow Ready\text{-}to\text{-}add.j$

$SWP3 :: (readyToRemoveCount.swp.j = numCount.cmp.j) \rightarrow Ready\text{-}to\text{-}remove.j$

$SWP4 :: (\neg readyToRemove.j) \rightarrow$ allow token to circulate. Allow resource access accordingly;

$SWP5 :: (readyToRemove.j \wedge holdsToken.j) \rightarrow removalApproved.j = true;$

$SWP6 :: (readyToAdd.j \wedge removalApproved.j) \rightarrow$ circulate $readyToRemove.j$ before token. allow token to circulate. Allow resource access accordingly; set $removalApproved$ to true at next component in new sequence.

Lemma 4.2.2 Removal Approved corrects the replacement of the old protocol with the new in the above program.

Proof: Safety follows from the fact that that the new component is allowed access to the token only after the old one has specifically removed and stop circulating its token in SWP5.

Liveness follows from the following:

1. SWP0 assures that at least one component will have its $readyToAddCount$ and $readyToRemoveCount$ set equivalently to its $numCount.j$, since every component will eventually receive the pseudo token.
2. SWP1 assures that eventually every component will have these same values.
3. SWP2 assures that eventually every component will be ready to add the new component.
4. SWP3 assures that eventually every component will be ready to remove the old component.

5. SWP5 assures that exactly one old component will relinquish the token when this occurs.
6. SWP6 assures that the old token may be removed everywhere and that the new one will circulate everywhere.

Stability follows from the fact that no action enables token circulation on the old component once it has stopped, and no action disables circulation of the token on the new component once it has begun.

If there are particular initialization or finalization concerns, these predicates may be added to be certain these are met before the new protocol is started, or the old one is removed.

4.2.4 Motivated leader election

Many leader election protocols, for instance the Bully Algorithm [11], or the one used by Arora and Gouda in [12], are based on a total ordering of process id's. We contend the following three conditions are likely to hold in a significant number of instances:

1. It may be desirable to choose a leader based on a specific set of criteria.
2. There are significant and stable differences among components with regards to these criteria.
3. It is not always possible to determine whether or not one component is better suited than another according to these criteria.

In short, some candidates for the leader position may be better than others, yet it is not always feasible to tell which one is best. For example, in a sensor network, it may be very desirable to select a leader that is either located near a certain location, away from physical obstructions, or near an item of interest, yet we may only be able to approximate values for any of these criteria. We will approach the problem by observing that the concatenation of a partial ordering followed by a total ordering yields a total ordering. Lamport uses this to construct vector clocks in [13]. We also show that given a set of components, each of which is able to provide some value with respect to a given criterion, we can construct a partial ordering on the values provided by those components. For any pair of components i and j in a system meeting the above criteria, we can construct detectors with the following predicates:

LE detector 1a ::

Z detects $i > j$,

and its converse

LE detector 1b ::

Z detects that $j > i$

Where $i > j$ denotes that i is better suited to assume a leadership role within the network than j is. By applying both of these detectors in parallel to these components, we will eventually arrive at one of the following conclusions:

1. $i > j$
2. $j > i$

3. these detectors cannot determine which component is better suited for leadership.

By applying the sequential composition of the atomic $>$ detector for integers on each process id to the parallel composition of this pair of detectors, we have a $>$ detector that operates on a total ordering of components within a system, defined formally as follows:

LE detector 2 ::

Z detects that $j > i \vee (\neg (j > i \wedge i > j)) \wedge (id.j > id.i)$.

As long as the safety property of the LE detector holds we are guaranteed that the ordering implied by this predicate is unique and provides as much information as these detectors can about which leader would be best. If the witness predicate for the LE detector is not stable, then the protocols will continually be in a process of re-election. If necessary, stability can be added to the detector by composing it sequentially with a timeout as follows:

Let Z be a predicate equivalent to the detector in LE detector 2 that acts on a copy of its variables. From this we can build a nonmasking LE corrector 1::

Lemma4.2.3 (*synchronizedtimeout*) \rightarrow update variables of $Z \parallel$ *LE detector 2* \parallel *Stabilizing (to node joins and failures) Leader Election based on $>$ corrects a single, fully-known motivated leader for U in any non-interfering program*

Proof: The synchronized timeout guarantees a sufficient amount of time when all components will create an identical ordering of events. A stabilizing program will work with this by definition, since it is designed to work from an arbitrary state once all faults stop occurring. After this point (until the next timeout), the algorithms will behave as though based upon constant values.

The action that updates the variables of the witness predicate after a timeout has occurred allows the detector to act exactly as its predecessor would, but only change its value after a certain time has occurred. This prevents minor, rapid fluctuations in the system from continually forcing re-elections while still maintaining a correlation to the motivating concern. The predicate of interest of *nonmasking LE corrector 1* is that if j is a leader, either j is a good leader, or its minimum term has not expired. Similarly, if j is not a leader, either j is not a good leader, or it has just lost an election and needs to wait for the next term. The witness predicate Z of *nonmasking LE corrector 1* verifies this specification. This technique can be used to extend the $>$ to suit a more meaningful purpose in nearly any stabilizing program, since temporary changes in the operation of the $>$ operator provide behavior that could also be modeled as a transient fault, to which stabilizing programs are defined to be able to correct.

We have shown from a theoretical standpoint that detectors and correctors provide a sound, robust and elegant mechanism to introduce tolerance components, which are also detectors and correctors into a running system in a fashion that is self-maintaining. We have also shown that they may be used to improve upon themselves

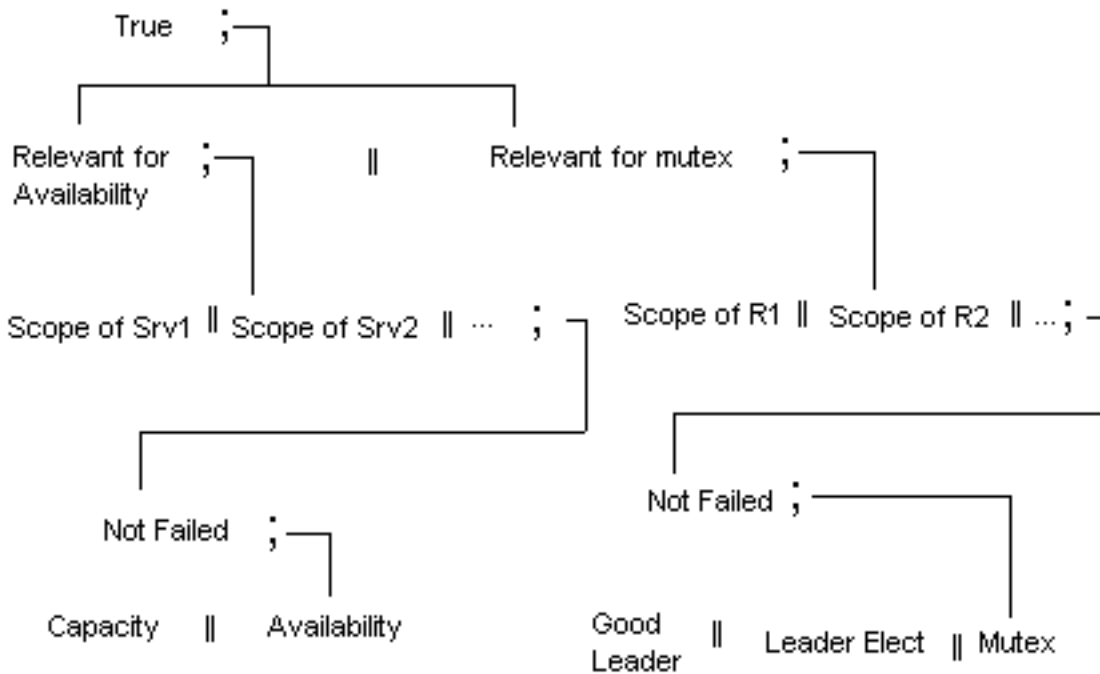


Figure 4.1: System Overview. Every component in our system model begins with the sequential composition of a detector for relevance and for scope. Virtually all components would also include a component that verifies whether or not they are up as well. The same mechanism used to define sequential compositions is used to guide the physical construction, destruction, and maintenance of components contained therein.

or even help replace themselves as preferable solutions are inevitably discovered. Figure 4.1 shows a system containing a set of the compositions we have described. We will now look more specifically at how components of this sort can be installed at run-time in a client-invisible fashion within a middleware and interceptor framework through a case study implemented in DRSS.

CHAPTER 5

OVERVIEW OF PROJECT GOALS

We wish to provide mechanisms and methodologies to improve the tolerance of a running, distributed system that require no modifications to the user programs in the system that are currently running, or potentially, that will be created in the future. We will deem a protocol to be client-invisible if it can be implemented without any changes to the client program. We will allow a component that is deemed client-invisible to execute methods on the underlying programs, provided that these do not need to be changed in any way. Put another way, if a component can be implemented in a general case by simply listening to and modifying the messages produced in a given system, then we will consider it to be client-invisible. This property potentially allows a component that has been developed and tested to be introduced into systems that are not prepared for its introduction, have not been re-compiled, or perhaps not even restarted. We also wish to develop components that may be accessed and manipulated at run-time in as natural and type-safe a manner as possible. Our theoretical target environment will be any system that uses any common middleware service, such as CORBA, RMI, or .NET remoting, but the specific environment for our work will be the DRSS framework created by Jason Hallstrom using the Microsoft .NET environment. We will show that a wide variety of protocols, which we will define as

resource-oriented, can successfully be implemented in a client-invisible fashion within such a framework. We will also show that there exists a large class of protocols, which we will define as peer-to-peer, that cannot necessarily be implemented in a client-invisible fashion within such a framework. We believe both of these findings to be major contributions of this work. We also show that there is a direct correspondence between the naming conventions that we use to identify and place individual components within compositions and named scoping conventions used by high-level languages, allowing for compositions to be viewed as typed, high-level objects.

5.1 Human roles within the system

We imagine two primary types of roles that a human will play in the types of systems we hope to influence within a detector/corrector framework. The first role is that of a user. The user will write code that uses the infrastructure, but does not necessarily take any measures to provide tolerance to the system, is not necessarily concerned with the consequences of concurrent accesses to resources, and focuses primarily on the application logic of the underlying programs. In particular, since a middleware service allows a sequential program to be run in a distributed setting, we can imagine a typical user to have created such a program, which was converted to make use of the middleware service later. The second role, the manager, provides the infrastructure to the user to allow the application to perform in a distributed setting, and creates and maintains tolerance components to assure that the user's view of the system is as close to a single process, sequential view as possible. For the purposes of this work, we seek to minimize the user's knowledge of the management system. In particular, the user only needs to know how to acquire a reference from

the system, and that invocations made on these references may fail. This requirement is present in any common current middleware service. The manager's task will be to eliminate these failures by introducing tolerance components which are invisible to the user. It has been shown [14] (and by virtually every software engineering project ever attempted) that the manager cannot be completely successful to this end. However, it has also been shown that under certain assumptions, the manager can provide a system which eventually meets this goal [1]. The basic mechanism that the manager will use to this effect is to intercept the messages that are sent by the client programs, and affect their behavior as necessary to provide clients with the most reliable system possible.

5.2 Project structure

Since the detector corrector framework is designed to allow code to be compiled and added to a running system and a library may not be recompiled while it is in use, the evolving framework will consist of a series of libraries each of which may reference any of the libraries previously established within the framework. In addition, we create a library to contain the components being controlled and an executable to instantiate all of the above. The breakdown of these projects in our preliminary implementation is as follows:

5.3 Enumeration of projects, libraries and roles

The **DRSS** project created by Jason Hallstrom provides the underlying infrastructure for connectivity of entities as well as the mechanisms to intercept messages.

The **DProcess** project contains a set of resources and consumers that will be monitored and controlled by the tolerance components.

The **DetectorCorrectors** project contains the building blocks for tolerance components, as well as the platform extensions used to facilitate predicate based maintenance of tolerance components.

The **DCLibraries** project contains specific instantiations of tolerance components and the mechanisms by which they are introduced into a particular portion of the system.

The **DGeneric** server contains the application that launches resources and consumers as listed above as well as the tolerance components that control their behavior.

CHAPTER 6

IMPLEMENTATION MECHANISMS, FINDINGS, AND IMPLICATIONS

We will now discuss the implementation of the project. We will summarize details of the DRSS framework on top of which we will build our solution. In building this solution, first, we will take steps to see that the information provided to us by the DRSS framework is in a form that is convenient for the particular types of components we would like to maintain and remark on some of some of the logical consequences of working within a middleware environment to achieve this goal. Next, we will discuss the structure of the framework, and how it is used to maintain tolerance components at runtime. Finally, we will discuss a case study implementation built with the framework.

6.1 DRSS

We will focus our attention first on the DRSS library, created by Jason Hallstrom. First it provides the ability to create remote references to objects in a fashion similar to other middleware services such as CORBA, Java RMI, and .NET Remoting and make calls with them as though they were local. In the DRSS framework (as with many other middleware systems) the references created to this end are referred to as

proxies and the objects being referenced remotely are known as targets. DRSS also provides the ability intercept messages as they are leaving the sender, just before they arrive at remote object, as they are returned from the remote object and just before they arrive at the sender. This is done with a series of chains of objects. The chains are identified as either send chains or receive chains, and the objects contained within them are referred to as interceptors. Each invocation from any sender or proxy goes through a send chain associated with it, the receive chain associated with the target, the send chain associated with the target and the receive chain associated with the proxy. These chains may be shared by multiple objects or proxies. This provides a basic infrastructure that allows a system manager to intervene with remote message calls in a fashion that is invisible to the user of these calls.

We will focus our attention next on the **DetectorCorrectors** library. This contains two primary namespaces, the *PlatformExtensions* namespace, and the *Guards* namespace.

6.2 Platform extensions

The DRSS framework is designed to allow for a wide variety of client-invisible component modification at run-time. We will use its capabilities to build our MDMC framework. As a first step, we will establish an item definition that provides a consistent interface to the set of items that connect to and are reachable from a target. From this point we will get an idea of what type of information is always available for tolerance component designers working at run-time, what types of information are not always available, and what strategies may be used to build tolerance components given this information.

6.2.1 Mappings provided

We will begin by discussing the role of the *PlatformExtensions* namespace. Common middleware services such as CORBA, RMI, .NET Remoting, or the one built into the DRSS subsystem generally seek to provide the illusion of a local reference by means of a proxy that forwards the call to another machine. They do this by packaging the arguments and method name in a well known format, sending the message with a unique id to a target identified by a particular address, and packaging and sending the return values with the id back to the proxy's address. The calling message itself therefore does not need to contain any information about what the type of the target is, what other entities may have a reference to this target, and what path the message must take to reach its destination. For the purposes of this project, the path we are interested in consists of a set of interceptors associated with the target and proxy known as send and receive chains. We then create and maintain a mapping from each target to its users and the paths between them and vice-versa. While the co-ordination of all proxy and path information associated with a single target is not necessary to forward individual messages to it, this information can be extremely helpful when deciding what actions to take to correct or detect conditions of interest within any given subsystem of a given system. The mapping also provides a convenient mechanism to affect all references to a given target once a system is running even if we did not anticipate referring to this collection of entities as the system was established.

The client-invisible information that is available within the system is analogous to what can be extracted from other systems that attempt to make distributed calls look like local calls and leads us to make very strong claims as to what can be done

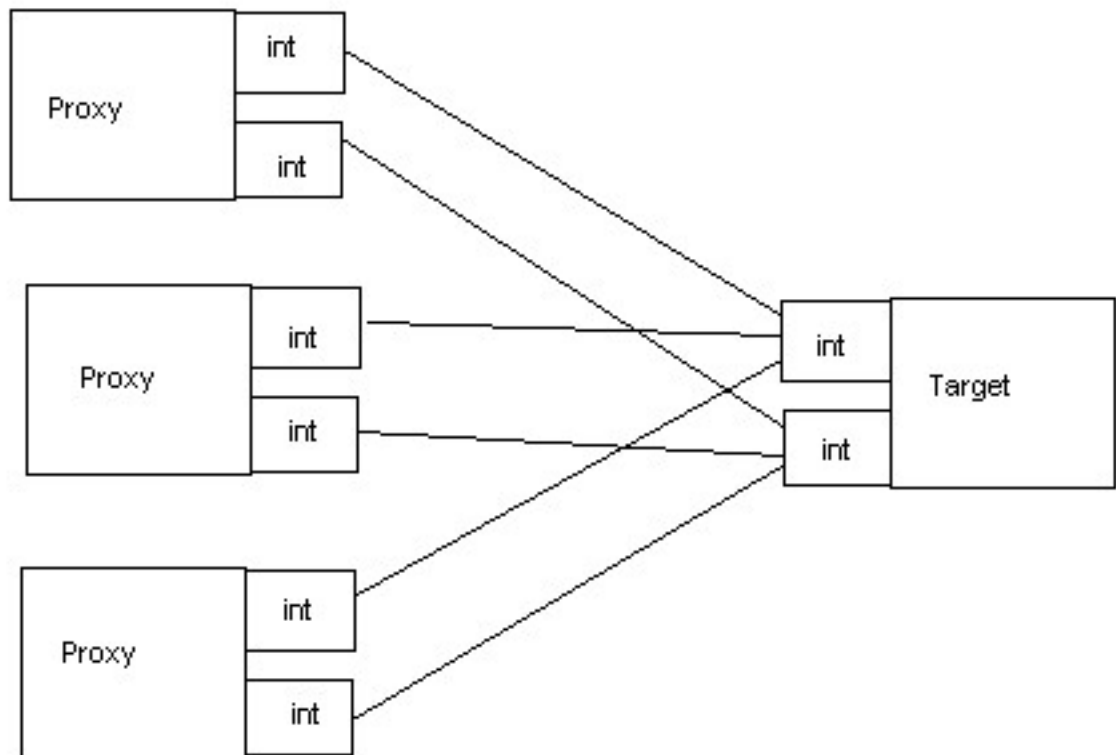


Figure 6.1: Diagram of a client-invisible tolerance component's view of the system. Tolerance components reside within the interceptors and can view the target, each other, or the proxies that reference the targets.

in a client-invisible fashion in this type of system and also what cannot be done in a client invisible fashion. These claims are dependent upon the implementation of the middleware and therefore are not proofs, however, they reliably hold until there is a shift in the state-of-the art paradigms for middleware, and not without good reason.

6.2.2 Assertion of capabilities

The information necessary to create targets and proxies that provide middleware-style references to remote targets is sufficient to provide client-invisible resource-oriented management. This management capability includes any calls that may be made on the target without adversely affecting the client, as well as any modifications to the status or behavior of proxies of this target, any changes to messages received by or sent from the target at either the client or server side of communications.

Justification:

The registration process that a client must follow to create a proxy involves knowing or finding the address of the target and assigning one to the client. This provides enough information for a management entity to create its own proxy for the target, thereby allowing it access to the target object while maintaining client invisibility.

6.2.3 Assertion of impossibility

The information necessary to create targets and proxies that provide middleware services is not sufficient to provide peer-to-peer services in cases which the middleware service needs to know about both peers simultaneously. This implies for all practical purposes, that peer-to-peer protocols which require some information about the underlying objects cannot be created in a client-invisible fashion within a common middleware service, and is particularly an issue for problems requiring state

detection/correction. For example, if a detector in a middleware service is able to determine that the sender of a given message must be corrupt, it can isolate the proxy that this sender used to send a message, but does not necessarily have any mechanism to affect any other proxies this sender may hold (without affecting the entire system) and does not necessarily have a method to ask the sender to reset itself even if the sender has a well-defined method to do so. It is also not generally possible for a Boolean predicate posed by a client program to be updated so that it verifies the *inst* [9] property as defined by Cooper and Marzullo for generic distributed predicates. Furthermore, deadlock detection and non-blocking distributed snapshots also require knowledge of the relationship between client and proxy, as well as access to client state, and therefore can not always be implemented in a client-invisible fashion.

Justification:

When a proxy is created to reference a target, it is given its unique address so that the call may return, but it does not require a handle on the object that created it in order to be able to make calls on the remote target. This is directly analogous to the way function calls work in sequential programs, and does not pose any potential surprises from a security standpoint. In fact the peer that created this reference may not be set up as a remotely accessible target at all. This implies that when a message is intercepted, regardless of the bookkeeping added to the middleware system, there is no guaranteed client-invisible technique to identify the object that holds the proxy that forwarded this message. For example, a client posing a generic distributed predicate will almost certainly use several proxies, none of which can necessarily be linked to each other or to the client. This prevents the system from knowing if it must

falsify a given response in order for the response of the whole to be conservatively correct.

6.2.4 Corollary to assertion of impossibility

To be able to provide any form of tolerance supported by the accessible structure of an underlying program in a client-invisible fashion, it is necessary and sufficient to provide a mapping between the proxies owned by an object, and the object that owns them.

Justification:

From assertion 6.2.2 we know that we can invisibly construct a management service that is able to access the set of users of a target as well as the target itself (in addition to other management components). The mapping of the users of a target to the client that created them provides full access to all of the components in a middleware system as well as the relationships between them. When a message is sent, it is now possible to identify the client that owns it, which allows for a protocol that depends on information from the client or the ability to affect the client to be implemented. Note that this capability is significantly limited by the amount of state that can be retrieved and/or modified by owning references to the clients and targets. In our case, public and private state is available, but local variables and parameters are not.

6.2.5 Mitigating factors and workarounds

Although we have shown that it is not always possible to create a mapping between a proxy and the client that owns it, there are cases in which this is possible. Some middleware services may provide a mechanism to build this mapping. For example, there are cases in which only one client component is active at any given socket. In

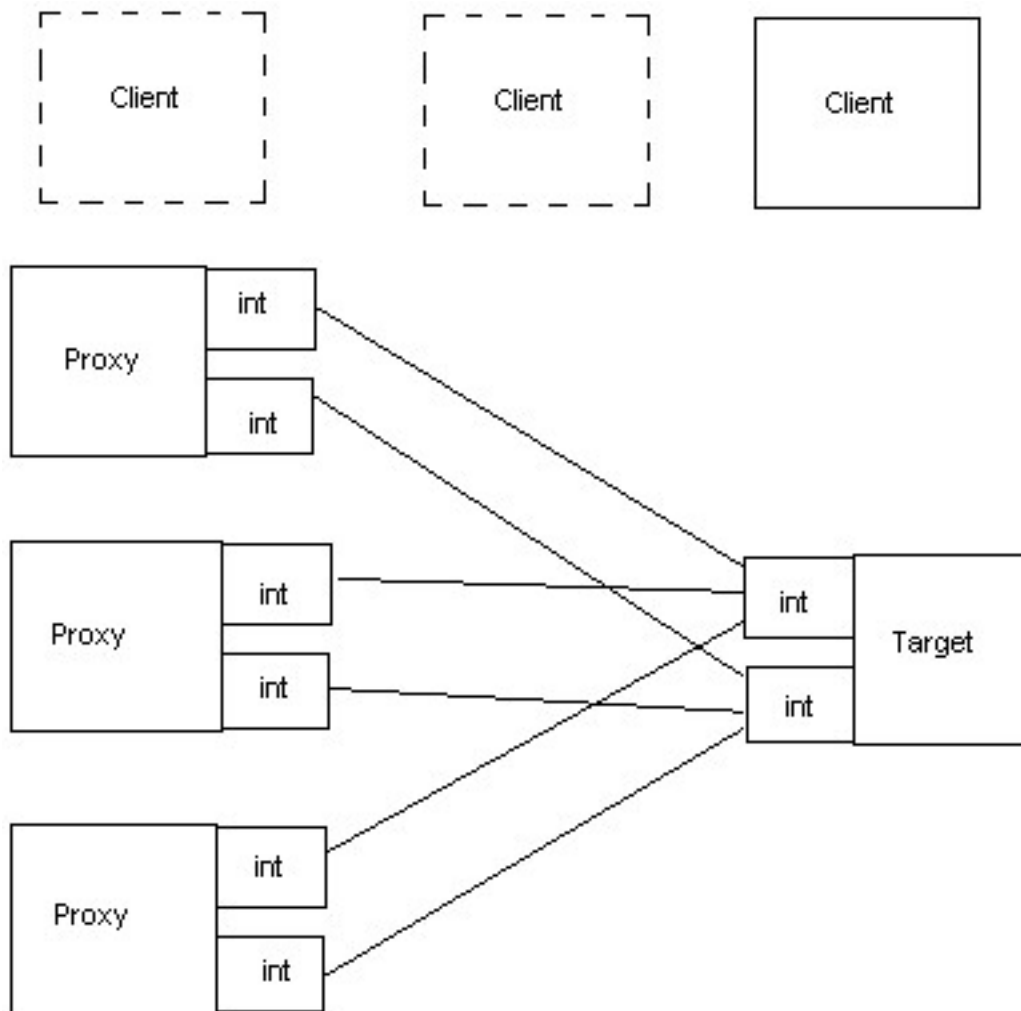


Figure 6.2: Diagram of a client-invisible tolerance component's view of the system with clients added. Although tolerance components can view the target, each other, and proxies, they do not necessarily know anything about the relationship between clients and proxies. The target in this diagram may be referred to by 1, 2, or 3 clients. Any or all of the clients may be the target itself. It is this asymmetry of knowledge that allows components to affect how a target is accessed, but not necessarily be able to control the behavior and/or view of clients.

such a case, the socket identifier can be used to create the mapping between the proxy and the client. In other cases, the client identifies itself within the contents of every message it sends. In cases where the client knows it is performing a set of actions that must be executed atomically, it explicitly indicates that these should be part of a transaction, and provides an identifier for that which is accessible to the middleware layer. Finally, with a fairly minor requirement of change at the client, namely that the client announce itself to the middleware service as elements are registered, we can build and maintain the client-proxy mapping in all cases. With the use of macros, aspect-oriented programming, or custom attributes, it may be possible to add this behavior to many client programs with compiler-generated rather than hand-written code changes.

6.2.6 Overview of tolerance capabilities and client visibility

Even if the tolerance components have no knowledge of the semantics of an underlying system, certain components may be added to increase the reliability of a system. Sliding window protocols may be employed to increase the reliability of message delivery in such a system [10]. Distributed caching can be used to statistically decrease the amount of message traffic sent and potentially increase message availability. Logging services can be added which may greatly facilitate the debugging process if an error is present within a system. If a system provides a service to save its entire state on a current machine, then a middleware service can use this to collect distributed snapshots even in the event that no application-level knowledge may be available to the middleware service.

Semantic application information, including possibly the ability to view the underlying source code can be used to increase the set of tolerance components that may be added to a system. This angle has been pursued by Pascal Felber et al in [15], under the assumption that the message information in conjunction with semantic knowledge of the program can be used to determine when certain types of tolerance may be required and certain approaches that may be used to enforce them. In particular, if rules can be formulated regarding access to a given target, these may be used to guide caching, replication, and certain mutual exclusion services.

In fact, many cases of the mutual exclusion requirement are concerned with a series of reads and writes to multiple objects. In such a case, a program which was designed sequentially, but has been modified for parallel use may contain *logical transactions*, or sets of statements whose behavior is not guaranteed to be correct if they can be interrupted by another component or the resources they use may be modified. These may require knowledge of the client-proxy relationship as well as program semantics, message and target information to determine and/or enforce the required tolerance. For instance, deadlock detection requires a specific knowledge not only that a resource is being accessed, but also by which client it is being accessed. Since a deadlock inherently involves multiple resources, it also involves multiple proxies, and a middleware service is not guaranteed to be able to know that these belong together. Therefore, enforcing logical transactions without potentially causing deadlock requires client knowledge. Only slightly more client knowledge is required to allow for partially non-blocking generic predicate evaluation. In such a case, it is necessary to replicate the entire predicate within the first interceptor one of its messages passes through. Still other protocols may wish to reset the state of a client in the event it is sending

corrupted messages. This can only happen if it is possible to associate the message with client, as well as the proxy that sent it.

Finally, there are many cases where very sophisticated knowledge of the underlying application is required. A middleware service does not generally have sufficient information to reproduce the current state of the program call stack, local variables and locally declared heap variables, even if the middleware has access to the public and private state of all clients and targets. Snapshot algorithms, such as Chandy-Lamport's [16] or variants such as Lazy Snapshots[17] require this type of knowledge be saved when the state of a process is saved. If system-level primitives exist and are accessible to assist a middleware service to achieve this end, they are likely to be extremely expensive as system state includes much more information than what is relevant for a given application. Application level snapshots, such as proposed by [18], avoid saving the entire system state, but maintain all of the information necessary to reconstruct the call stack, global, local, and heap state of the components they wish to be able to restart. This makes use of significant, compiler-like, preprocessing to store the extra state mentioned in such a way that it is available at the middleware layer. Some transaction executions and/or distributed predicate evaluation necessarily involve knowledge of local variables, and therefore require the ability to access this state as well as local client state. We note here that in the special case that all procedure calls are synchronous, and the underlying system's invocations of each object are mutually exclusive, the Lazy Snapshot procedure can be used to take consistent non-blocking snapshots without requiring any access to local state or the call stack. In these conditions, the previous level of client awareness is sufficient.

Table 6.1 summarizes what types of tolerance components may be added to a middleware system given different levels of involvement with the underlying components.

6.3 A system for creating programs, modifiable at run-time with high atomicity - programs, components, and guards

We will now discuss the particular components from which we will build and maintain tolerance components at run-time. These components are modeled after the guarded command paradigm, but have additional identifying information associated with them to facilitate run-time access to them.

6.3.1 Programs

While we maintain the appearance of single-process, synchronous function calls for the client programs in this project; we also wish to be able to make detections and corrections which may be asynchronous in nature. Onto each send interceptor chain and receive interceptor chain (one per proxy and one per target) we have the ability to attach a program, to which many components may be added, each of which contains a number of actions. The program is roughly equivalent to a SIEFAST program using guarded commands with the following perceived differences:

1. It can be added to a system that is already running
2. Components (containing new actions) can be added to the program while it is running
3. Guards may be strengthened or weakened while the program containing them is running.

4. A subset of actions, labeled synchronous actions, are scheduled simultaneously when a message is received from the underlying user program, rather than individually according to the scheduler.

This combination of capabilities allows us to perform composition of detectors and correctors while the underlying program is running and also while the detector or corrector itself is running. This can be particularly useful if it is necessary to hot-swap one component with another.

6.3.2 Program components

The program serves two primary functions. The first is to provide an attachment point for program components. The second is to schedule asynchronous actions within the program components. The component contains the state it needs to operate and is responsible for adding its actions to the scheduling program. Program components are added to programs using setup objects that contain a string identifier, capable of uniquely identifying that component within the space of the program, and set of arguments that customize the component to fit the particular underlying program or other component to which it is being attached. In this way one component may be added to different programs if necessary to fulfill multiple roles.

6.3.3 Actions

In the same way, actions may be added to programs after they have been started, by using the *AddAction* method of the program. Should the new action require access to the state of any of the existing program components in the program, it can obtain a reference to the component using the identifying string that was originally used to add the component to the program.

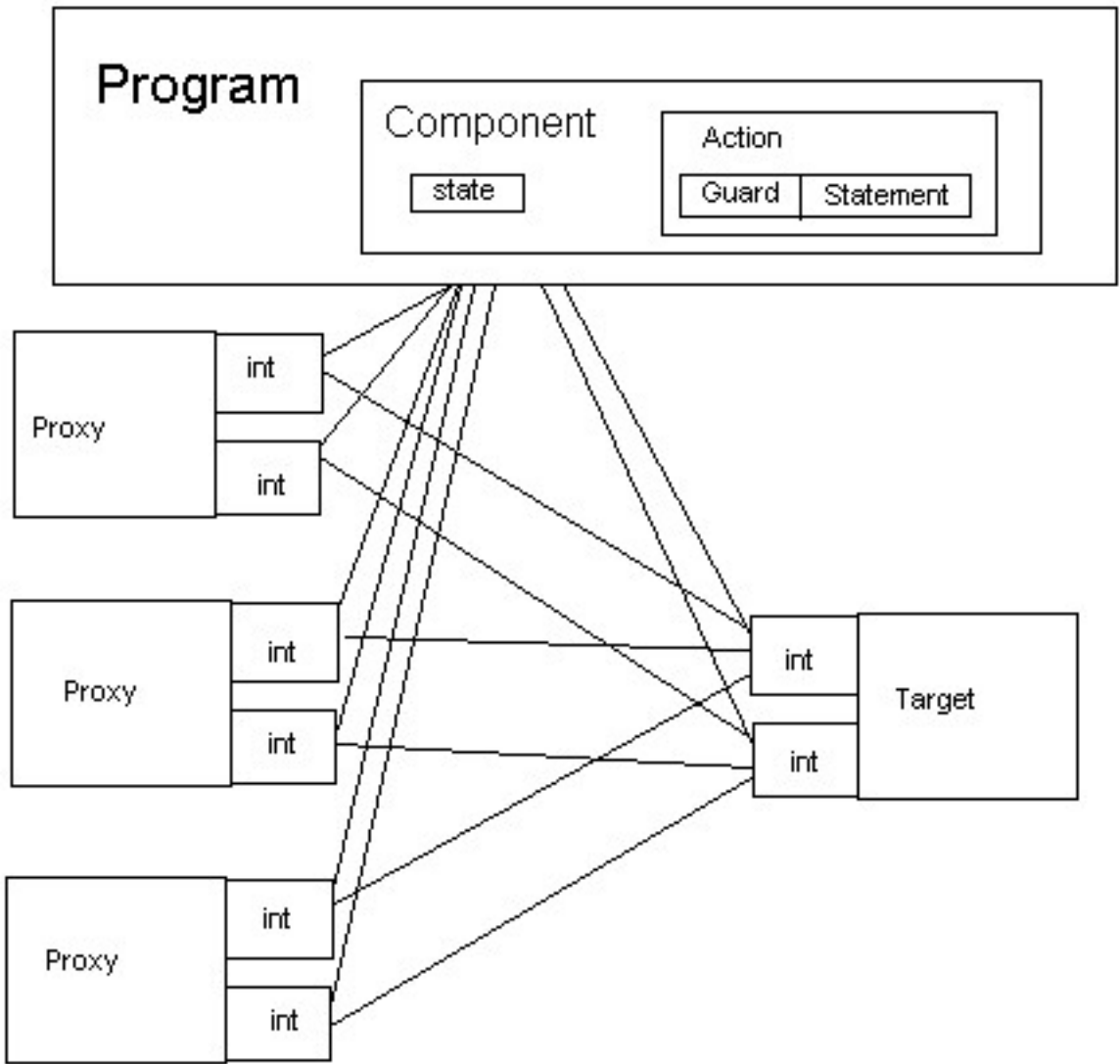


Figure 6.3: Diagram of the set of programs, program components, and actions within a system. A program may be attached to any interceptor. In turn, it may contain any number of components, each of which contains actions and state. Actions contain guards and statements. The guards of any of the actions may be strengthened or weakened at run-time, more actions can be added to a component, and more components to a program. Newly added program components have full access to the state of previously added components. In this way, components may be composed at run-time and the total system impact can be limited to the set of actions whose guards must be strengthened for safety purposes and the overhead of loading and running the new components.

Finally, each action is itself labeled, so that it may be uniquely identified within a program, and its guard strengthened or weakened as appropriate when program components are composed, using the *AddCondition* method, that takes a boolean function, identifying string, and operand (and or or) for its arguments. The resulting function is either the conjunction or disjunction of the existing guard function and the new one accordingly, and is now named by the new identifying string. These actions may be scheduled in one of two ways, asynchronously or synchronously.

Asynchronous actions are scheduled by the program's scheduler, and run independently of any other events occurring in the system. One action is selected from the list, its guard is evaluated, and if true, the statement is performed.

Synchronous actions are scheduled whenever the *InterceptMessage* method of the interceptor attached to the component is called. For these actions, the guards are evaluated in parallel, each guard is evaluated, and, after all of the guards have been evaluated, the corresponding statements for every guard that evaluated to true are executed. This is similar to the input action paradigm of Nancy Lynch's IO Automata [19] whereby all input actions are fired when a corresponding output action is fired, and is used for all actions that either can or must be executed as soon as a message arrives.

In this way program components are able to perform a wide variety of actions that occur either as a result of system events or their own scheduling, and either of these types of actions may be updated while the system and components are running.

6.3.4 Resulting capabilities

The ability to modify the actions and state of running programs in the above fashion allows direct support for the types of actions that are required for the composition of detectors and correctors to be performed at run-time. This means that we can plan system changes according to the properties of these compositions, while only affecting a small portion of the actions involved. As we demonstrated in the case of component swapping, we can even get different components to do this themselves. From a programmatic point of view, the convention of naming, retrieving, and modifying components according to identifying strings that are unique within the program space allows us to not only reuse lower-level components in different programs within the same system, but also to reuse their names. So if component A and component B were refinements of different sequential compositions (in our case, programs, but this will be generalized in future work) and had no properties or subcomponents named up for example, this component could be added to either or both at run-time. Furthermore, corresponding subtypes could automatically be created that safely referred to $A.up$ or $B.up$ respectively. For instance, if we wanted to change the tolerance of a given program from non-masking to masking, we could add a condition containing the safety predicate or a witness to it to all of the potentially unsafe actions as described by Arora and Kulkarni in [20].

6.4 Installing and maintaining tolerance components according to relevance and scope predicates - protocols, protocol groups, and templates

We have seen how we can add functionality to a single program, attached to a send or receive interceptor on a single target or proxy. In all likelihood, more interceptors

will be required to provide additional tolerance to a given system. In the first place, certain actions may be required when the response to a message is returned to the sender as well as when the message itself is sent. Secondly, many proxies and/or corresponding targets may need to run the same program or set of programs to add the desired tolerance properties to a system. We define a protocol to be a set of programs working together to add a desired tolerance property to a particular target to proxy relationship, a protocol group to be a set of protocols addressing a common concern across a set of components within a system, and a template as a mechanism for locating the members of a protocol group within a system.

6.4.1 Templates

We will begin with the template, because it provides the foundation for not only how, but why the other two entities exist. The template is simply an object that takes a second in as an argument, and applies a Boolean match function to that object and a target item. As discussed above, the target item contains information to view and/or modify any information about the target, as well as a mapping from it to the proxies that reference it, and any tolerance components attached to itself or its proxies. This allows a manager to inquire about a target's state, the number of users it has, and the set of tolerance components that have thus far been added to it or its proxies. In short, this provides the manager a means to express the relevance predicates defined earlier in this paper. Each platform has the capability of presenting this inquiry to all connected platforms within a system, thereby locating the set of target items that match a given concern.

6.4.2 Protocol groups

A protocol group is responsible for maintaining a list of all target items of interest within a system, and assigning each to its appropriate protocol, maintaining a list of protocols as well. The group of items is the relevance group as defined earlier in this paper and it is the job of the protocol group object to detect whether any item in the system is a member of this group, and if so, apply the necessary detection or correction protocol to it. The default implementation of the group is first to locate all relevant entries in the system in a sequential fashion, and subsequently, to detect any system additions which may be relevant to the group, but a variety of detectors and correctors may be added to this group in order to make this determination more appropriately match that of the ideal detector.

6.4.3 Protocols

A protocol is responsible for maintaining the list of items that meet the scope requirements. In many, by no means all instances, this would be the set of proxies that address a target of interest. It is the protocol that installs the programs and components necessary to add the desired tolerance to a system and attaches them to the appropriate interceptors of interest. It is responsible for knowing what components need to be installed for the system to work, how to link them together so that they achieve their common goal, and maintaining the list in order as new candidates are added and/or removed. Detectors and correctors may be combined with it in the event that the criteria for membership change over time.

We have seen that detectors and correctors can be used to perform a wide variety of tasks in a running system, facilitating their own instantiation and maintenance.

We will now look at a specific example of the system being used to create a tolerance component that is designed independently of the underlying system components it affects. It adds tolerance to an intolerant system at runtime, and continues to provide tolerance as the relevance and scope of domain of this type of protocol expands within the system.

6.5 Case study : dynamic token ring

We begin with a scenario that we have an underlying system that contains two types of objects of interest, namely resources and consumers. We further postulate that the resources were not originally intended to be used in parallel by more than one consumer, and that each individual consumer accesses the resource in a sequential fashion. We decide to implement a protocol so that consumer access to any resource occurs in a one-at-a-time fashion. Furthermore, due to the particular nature of the system, it is preferable to negotiate consumer access at the consumers as opposed to the resource for this particular application.

6.5.1 Identifying and connecting to relevant underlying system components

To begin with, we'll want to gather up the resources of interest along with their users, grouping the users with the associated resources. This is done by creating a resource template, and passing it into the constructor along with two other arguments, a method name that the consumer program uses to acquire the resource, and one it uses to release the resource. If the resource access consists of a single method call, the user simply provides the same name for both accessing and releasing the resource.

6.5.2 Controlling resource access

If the component does not believe it owns the token when the message is received from the consumer it simply holds the message and waits, periodically checking to see whether or not it has the token, and forwarding the message when it does. Once it receives the response from the release message, it releases and forwards the token to the next waiting consumer. The protocol is very similar to Dijkstra's token ring or the message passing version presented by Afek and Brown [10], but there are a few more states to accommodate the message passing mechanisms of the interceptor framework. The actions of the program are summarized below. (Synchronous actions will be marked sync. These actions are all scheduled every time a message is received but not at other times). For ease of exposition we have left out the specifics of message handling, address manipulation, exception handling and the like.

Sync1 :: ($msg = acquire$) \rightarrow store message, $mw=true$;

Sync2 :: ($msg = release$) \rightarrow store *releaseId* as not released;

Sync3 :: ($repMsgId = releaseId$) \rightarrow $holdsToken=false$; $curSeq = inSeq$; set *releaseId* to released;

Async1 :: ($isLeader \wedge curSeq = inSeq \wedge \neg holdsToken \wedge mw$) \rightarrow *SendMessage*; $holdsToken=true$; increment seq;

Async2 :: ($isLeader \wedge curSeq \neq inSeq \wedge \neg holdsToken$) \rightarrow $cohort.InSeq = this.curSeq$;

| Knowledge of client | General application access provided | Types of protocols that can be implemented |
|--|--|--|
| None | Message destinations but no semantics | Logging. Reliable-delivery. Process-level snapshots (assumes system level services and no more than one client per process). |
| High-level semantics | Message and target information | Above plus some mutual exclusion and load balancing |
| Source-code level semantics (no underlying code changes are required) | Access messages, know their meanings, know & access the destinations of messages & resources and can view, but not modify source code. | Above plus some mutual exclusion, load balancing. |
| Client-Proxy mapping (requires small source code changes in general case) | Access messages, senders, & receivers | Above plus client state visibility and manipulation and distributed non-blocking atomic predicates. Deadlock detection and most logical transactions. Special cases of distributed non-blocking snapshots. |
| Application-level stack (requires introduction of a small compiler to preprocessing phase) | Access messages, senders, & receivers, the application's call stack & local state | Above plus application-level distributed snapshots, complex logical transactions involving nested structures. |

Table 6.1: Varying degrees of involvement with the underlying system and types of protocols that may implemented at that level. Client-invisibility is maintained for the first two levels. A client-proxy mapping requires changes only to the interface between the middleware and its users. Protocols which require access to the application-level stack most often require complex preprocessing and/or modification of the underlying system.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this chapter, we discuss the findings of the project, and the applications of the material to future work. Much has been achieved to unify modern high-level programming constructs with state-based tolerance components, to form a coherent, yet separate whole. Much needs to be done to make these components more widely usable, reusable and efficient. We will investigate our experiences and explore future areas for growth based upon them.

7.1 Conclusions

The system handles the dynamic nature of the theoretical model. Whether the tolerance component is installed before most of the system components or at the very end, the results are equivalent; accesses of components deemed relevant that had been allowed to occur in parallel are brought in line with the rest of the system and controlled sequentially. Accesses of components not deemed relevant are left unaffected. We have a highly dynamic framework, which can be used to create components that may be reused and composed at a very granular level in a running system. We have shown that the predicate-based design allows for a naturally adaptable system and the detector-corrector formalism allows us to reason about the correctness of a

dynamic system in a piecewise fashion, simplifying the task. We have shown that the composition of detectors and correctors provides a natural mechanism to provide them with further ability to adapt themselves not only to their environment, but also their task as these may change at runtime. Furthermore, we have isolated and defined a boundary defining the types of components that can and cannot be created in a client-invisible fashion within the interceptor bounds of a middleware framework. Lastly, we have defined a relatively small violation to the principle of client-invisibility that allows us to acquire complete control over a client's public and private state, and identified the level of preprocessing necessary in order to access local variables and parameters.

7.2 Future work

Despite these successes, much work remains to make the system more efficient, user-friendly, and powerful. The introduction of strong, flexible visualization and testing tools, possibly including a model checker, will be essential for the development of evolvable reliability components within an evolving environment. To afford the full flexibility of run-time component maintenance, we propose the introduction of client-awareness into a middleware layer upon which a detector-corrector system is built, extending the potential for reusability of many peer-to-peer style components. The extension of this client-awareness to include local variables and (locally populated) call stack information of both clients and targets introduces a large degree of preprocessing into the system, but also provides greater ability to study and control system behavior. To meet the expectations of the synchronous system inside of which this is built, we have created synchronous action lists as well as asynchronous actions. We believe that

tolerance and simplicity may both be served by modeling all of these as asynchronous actions, but attaching some to specified system events for immediate scheduling. New iteration mechanisms are advised for composition objects whose membership is likely to change and or be changed as the objects within compositions are traversed. We also note that this system is primarily designed to work once a designer knows roughly what may require correction within a system and what conditions need to be true once that correction has taken place. Discovering problems and their root causes is likely to be a much less deterministic process where anomalous and/or suspect patterns of system behavior are brought to the attention of human beings that examine and make sense out of the data. This type of research does not necessarily lend itself to the pattern of specifying a predicate that should be true when investigation is started and specifying another that you know will be true when the investigation is complete.

Finally, as middleware technologies grow, a variety of pub-sub and other services are emerging that provide different types of guarantees than those of traditional middleware systems, many of them less synchronous and less deterministic. While it seems that detector and corrector components are a natural approach to queries that may not demand immediate and/or perfect information, there may be modifications to their interfaces that make them better able to respond efficiently to less deterministic queries. The guarantees provided by detectors and correctors may also be positively influential on the types of high-level interfaces that future middleware services provide.

BIBLIOGRAPHY

- [1] V. H. Tushar Deepak Chandra and S. Toueg, “The weakest failure detector for solving consensus.,” *Journal of the ACM*, vol. 43, no. 4, pp. 685–722, 1996.
- [2] V. K. Garg and J. R. Mitchell, “Implementable failure detectors in asynchronous systems,” in *Proc. Foundations of Software Technology and Theoretical Computer Science*, 1998.
- [3] Garg and Waldecker, “Detection of weak unstable predicates in distributed programs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 3, pp. 299–307, 1994.
- [4] V. K. Garg and B. Waldecker, “Detection of strong unstable predicates in distributed programs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 12, pp. 1323–1333, 1996.
- [5] S. D. Stoller, “Detecting global predicates in distributed systems with clocks,” *Distributed Computing*, vol. 13, pp. 85–98, Apr. 2000.
- [6] A. Arora and S. S. Kulkarni, “Component based design of multitolerance,” *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 63–78, 1998.
- [7] S. S. Kulkarni, J. Rushby, and N. Shankar, “A case-study in component-based mechanical verification of fault-tolerant programs,” in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems* (A. Arora, ed.), (Austin, TX, USA), pp. 33–40, IEEE Computer Society Press, 1999.
- [8] S. S. Kulkarni and A. Arora, “Compositional design of multitolerant repetitive Byzantine agreement,” *Lecture Notes in Computer Science*, vol. 1346, pp. 169–183, 1997.
- [9] R. Cooper and K. Marzullo, “Consistent detection of global predicates,” *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, vol. 26, no. 12, pp. 167–174, 1991.

- [10] Y. Afek and G. M. Brown, “Self-stabilization over unreliable communication media,” *Distributed Computing*, vol. 7, no. 1, pp. 27–34, 1993.
- [11] H. Garcia-Molina, “Elections in a distributed computing system,” *IEEE Trans. on Computers*, vol. 31, no. 1, pp. 48–59, 1982.
- [12] K. V. Nori and C. E. V. Madhavan, eds., *Distributed Reset*, vol. 472 of *Lecture Notes in Computer Science*, Springer, 1990.
- [13] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [14] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” in *Symposium on Principles of Database Systems*, pp. 1–7, 1983.
- [15] P. Felber, B. Jai, M. Smith, and R. Rastogi, “Using semantic knowledge of distributed objects to increase reliability and availability,” 2001.
- [16] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed system,” *ACM transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.
- [17] N. Sridhar and P. A. Sivilotti, “Lazy snapshots,” in *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems* (S. Akl and T. Gonzalez, eds.), (Cambridge, MA), pp. 96–101, IASTED, ACTA Press, November 2002.
- [18] K. P. Greg Bronevetsky, Daniel Marques and P. Stodghill, “Automated application-level checkpointing of mpi programs,” Tech. Rep. TR2003-1891, Cornell University, 2003.
- [19] N. Lynch, *Distributed Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers, 1996.
- [20] A. Arora and S. S. Kulkarni, “Designing masking fault-tolerance via nonmasking fault-tolerance,” *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 435–450, 1998.