THESIS
1
2003
54024935

This is to certify that the
thesis entitled

Dynamic Composition of Distributed Components

presented by

Karunkumar N. Biyani

has been accepted towards fulfillment
of the requirements for the

M. S. degree in Computer Science

[illegible]
Major Professor's Signature

May 2 2003
Date

DYNAMIC COMPOSITION OF DISTRIBUTED COMPONENTS

By

Karunkumar N. Biyani

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2003

ABSTRACT

Dynamic Composition of Distributed Components

By

Karunkumar N. Biyani

The thesis focuses on the development of distributed applications that can add and remove *distributed components* in order to adapt to environment conditions. A distributed component means that the component consists of several *component fractions* and when adding a distributed component to an application, each process in the application is composed with one fraction of the distributed component. The addition and removal of such distributed components introduces new challenges. Specifically, it is necessary that the addition and removal satisfy the following three properties: (1) *atomicity*, i.e., if a distributed component is added (respectively, removed) then all of its component fractions are composed with (respectively, removed from) respective processes; (2) *minimal blocking*, i.e., during the addition and removal of a distributed component, the application should continue to execute or any blocking introduced should be minimal; and (3) *synchronization*, i.e., while the component replacement is being performed, processes that use the old component should not interact with those that use the new component. The satisfaction of these properties is further complicated as the component fractions may depend on each other. Hence, the thesis develops a framework that supports the dynamic addition and removal of distributed components while dealing with the dependency relation among the component fractions. The thesis proposes such a framework based on the concept of distributed reset that focuses on resetting the state of a system to a given global state.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Sandeep Kulkarni for his guidance and support throughout this work. I would also like to thank Dr. Laura Dillon and Dr. Philip Mckinley for reading this thesis and serving on my committee.

I thank my colleague Umamaheswaram Arumugam for implementing the routing protocol in Siesta using the framework developed in this thesis.

Finally, I thank my family, my parents and my sisters, who were always there for me even from the distance. I am very grateful for their encouragement and support.

PREFACE

The software for the framework developed in this thesis is available at:

`http://www.msu.edu/~biyanika/software`.

If you have any questions or comments about the work presented in this thesis, feel free to contact me by e-mail at this address:

`biyanika@msu.edu`

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Separation of concerns is an important issue in evolving distributed applications. In its most general form, it refers to the ability to identify, encapsulate and manipulate those parts of an application that are relevant to a particular software concept, goal, task or purpose. Separation of concerns is one of the primary motivations for organizing and decomposing an application into smaller and more manageable parts. An appropriate separation of concern has been hypothesized to reduce the complexity of an application, improve comprehensibility, and promote traceability within and across the application components and throughout their lifecycle. Separation of concerns also facilitates reuse, adaptation, customization and evolution, and simplifies component integration.

We study the issues related to separation of concerns in the context of fault-tolerant distributed applications. We begin with the observation that in the initial design of many applications, it is difficult —if not impossible— to identify all the faults that may occur. Hence, during the evolution of a fault-tolerant application, we need to upgrade it to add fault-tolerance to new faults. Moreover, such addition needs to be done without rewriting the entire application. In other words, while adding new fault-tolerance, the existing application has to be reused. Furthermore, for a long-running application, this addition of fault-tolerance may need to be dynamic, i.e., while the application continues to run, we may need to add fault-tolerance to new faults.

In the context of fault-tolerant applications, we need to deal with their *functionality* and *fault-tolerance.* We propose an adaptive component-based framework that enables such separation based on the theory of detectors and correctors presented in [1]. In [1], it is shown that a fault-tolerant application consists of a fault-intolerant application that deals only with the functionality in the absence of faults and a set of fault-tolerance components that deal only with the fault-tolerance. These components are based on the concept of detecting (and restoring) *state predicates* (as opposed to faults or their symptoms) involving variables of the fault-intolerant application. In [1], it is also shown that these components are necessary and sufficient for a rich class of fault-tolerant applications that reuse the corresponding fault-intolerant application.

We also find that the components used in different fault-tolerant applications as well as the components used at different layers of an application are similar. For example, the detectors used in problems such as leader election [2], mutual exclusion [2] and termination detection [2] are similar. In each of these cases, the detector was used to detect predicates of the form "for each process *j*, detect if local predicate *local.j* is true." While the local predicate being detected varied from application to application, there was a significant overlap in the implementation of these detectors. Likewise, components used at different layers of an application are also similar. This suggests that techniques that can systematically reuse detectors and correctors are beneficial. It is also possible that for a given application, alternative detectors and correctors can be used to add fault-tolerance to it. For example, consider the problem of data transfer from one source to one or more destinations where messages could be lost. In this case, one could use a component based on the retransmission of lost messages or one could use a component based on the concept of *forward error* *correction (FEC)* (cf. Appendix A) where additional messages are sent up front. While either of these components can provide the required fault-tolerance, the choice depends on the environment, *e.g.*, rate of message loss, computing power associated with processes and application requirements.

Based on the above observations, we notice that to permit efficient reuse of fault-tolerance components and to separate the fault-tolerance concern, it is important to develop a framework that will support dynamic addition/removal/replacement of these components and their reuse. There are several issues that need to be addressed while adding these components. We discuss these issues in the following section. Subsequently, we discuss in Section 1.2 how our framework deals with these issues.

1.1 Issues in Adding Fault-tolerance Components

In this section, we discuss the following issues that arise while adding a fault-tolerance component to an intolerant distributed application: *independent development* of such components, *flexibility* of adding components; ensuring *correctness* after the components are added; addition, removal and replacement of components at run-time for *adaptation*; permitting such addition and removal of components that are themselves *distributed.*

Independent development. We expect that an appropriate and clean separation of fault-tolerance and functionality will help clearly identify the role of the developer who designs fault-intolerant applications and the developer who develops fault-tolerance components. Specifically, an application developer should concentrate only on the functionality and not be concerned about providing fault-tolerance. Also, the developer of fault-tolerance components need not be concerned about where these components are used. Such independent development of distributed applications should make it easier to manage and deploy them.

Flexibility and reuse. The component-based design of fault-tolerant applications poses new issues related to the integration of fault-tolerance components with an intolerant application. One of the important issues is the complexity of such integration. There should be no restrictions (or minimal restrictions) on the way the intolerant application is developed or the way the fault-tolerance component is developed. Moreover, the tasks involved in the composition should not be an impediment in using the fault-tolerance component. Such integration should be done at a high-level and that synchronization between the intolerant application and the fault-tolerance component be minimal. Further, this integration should be fast, easy and efficient. Automating this integration should minimize human intervention. The reuse of fault-tolerance components will be enhanced if we have fast and efficient mechanisms to integrate these fault-tolerance components with an intolerant application.

Interference and correctness. The integration of fault-tolerance components with an intolerant application may lead to interference that may hinder the intolerant application from satisfying its functionality or it may prevent the fault-tolerance component from providing the desired fault-tolerance. Hence, we need to ensure that the fault-tolerance components added to the application do not interfere with the functionality of the intolerant application and that the fault-intolerant application does not prevent the fault-tolerance components from providing the required fault-tolerance. Such interference freedom would ensure that the application would continue to function correctly upon composition. These correctness requirements have to be met, whether the components are added statically (at compile-time) or dynamically (at run-time).

Adaptation. For a given application, there exist several fault-tolerance components that provide the required fault-tolerance. In these circumstances, the choice of a component depends on the environment. Specifically, it is possible that while several components provide the required fault-tolerance, one component provides better performance than other components in a given environment. Thus, an application should be able to dynamically change a fault-tolerance component based upon its environment. Moreover, under such dynamic composition, the change of the component should be transparent to the user of the application. In other words, the user of the application should not be affected when the components are being changed.

Distributed fault-tolerance components. We often find fault-tolerance components in the distributed applications that are themselves distributed. A distributed fault-tolerance component consists of *component fractions*, one for each process in an intolerant distributed application. Under such circumstances, reuse, correctness and adaptation of fault-tolerance components introduce new challenges. For example, when a distributed component is added, multiple processes on multiple processors are affected. During this addition, if one process that has added its component fraction interacts with another that has not added its component fraction then the results will be unpredictable. Thus, we need to ensure that the addition of a fault-tolerance component is (1) *atomic*, i.e., the addition of a fault-tolerance component across the distributed application appears indivisible; (2) *minimally blocking*, i.e., during the addition of a fault-tolerance component, the application should continue to execute or any blocking introduced should be minimal; and (3) *synchronized*, i.e., two processes in the distributed application interact only if they are using fractions of the same component. Similar issues arise when removing or replacing a distributed fault-tolerance component.

1.2 Proposed Framework

Based on the issues discussed in Section 1.1, in this thesis, we propose a framework that supports dynamic composition of distributed fault-tolerance components while enabling independent development and adaptation.

The main feature of this framework is its ability to dynamically add, remove and replace distributed fault-tolerance components. More specifically, the framework ensures that the three properties, namely, atomicity, continuity and synchronization, are satisfied when a distributed component is added, removed and replaced. During replacement, we do not assume any relationship between the current component and the component being added. Moreover, our framework can be tailored to deal with faults that may occur when changing components. Our approach for ensuring these properties is based on distributed reset [3,4], which resets the state of a given system to a given global state.

Additionally, to ensure correct, flexible and interference-free integration of fault-tolerance components with an intolerant distributed application, we introduce the notion of *contracts*. These contracts are constraints imposed by the component on the intolerant application. It specifies constraints under which this component can be used. Satisfying the contract implies that the intolerant application adheres to all the requirements imposed by the fault-tolerance component. Intuitively, if the contract is satisfied, the fault-tolerance component will provide the required fault-tolerance when integrated with the given intolerant application. Our framework also includes an adaptation module that is responsible for providing adaptive fault-tolerance. Based on the *adaptation strategy*, the adaptation module chooses a fault-tolerance component for dynamic addition, removal or replacement.

Thus, our framework has the following features: (1) It enables dynamic composition of distributed fault-tolerance components to support adaptation. (2) It allows efficient reuse of fault-tolerance components. (3) It can be used as a middleware to provide fault-tolerance for faults occurring in a distributed application.

Organization of the report. The rest of the report is organized as follows: In Chapter 2, we describe the overall architecture of our framework and explain the functions of all the modules of the framework. We explain the reset subsystem and how it is used for dynamic composition in Chapter 3. In Chapter 4 and Chapter 5, and in Appendix B, we present examples where our framework is used. We discuss some of the issues raised by our framework in Chapter 6. Finally, we describe the related work in Chapter 7 and conclude in Chapter 8.

CHAPTER 2

Framework Architecture

In this chapter, we describe the architecture of our framework and explain each of its modules. In our approach, we provide fault-tolerance to an intolerant application by composing it with fault-tolerance components. The composition of fault-tolerance components with an intolerant application can be asynchronous, synchronous or mixed.

In asynchronous composition, the execution of the fault-tolerance component is not synchronous with the execution of the intolerant application. Typical examples of fault-tolerance components that require asynchronous composition are components that do the backup of a system, components involved in monitoring environment conditions, etc.

In synchronous composition, the fault-tolerance component is executed synchronously with the intolerant application, i.e., when the intolerant application executes some portion of its code, the fault-tolerance component needs to execute the corresponding portion of its code. We choose method-level synchronization for synchronous composition. Had we chosen a lower-level synchronization (e.g. statement-level), the complexity of composition would have been high. Lower-level synchronization is likely to be tedious to specify. Instead, method-level synchronization is expected to be efficient and easier to specify compared to statement-level synchronization. In method-level synchronization, the fault-tolerance component executes its method when a certain method of the intolerant application is executed. Towards this end, we trap the method of the intolerant application and pass control to the method of the fault-tolerance component. Depending on its implementation, the fault-tolerance component may call its own method and/or it may call the trapped method of the intolerant application. The *proactive component* (cf. Chapter 4) and the *AlternateRoute component* (cf. Appendix B) are examples of components that use synchronous composition. In case of proactive component, the decoding and encoding actions of the proactive component executes synchronously with the send and receive actions of the intolerant application.

Finally, in mixed composition, some piece of code is executed synchronously with an intolerant application and some piece of code is executed asynchronously. The *reactive component* (cf. Chapter 4) is an example of a component that uses mixed composition. In this example, the send and receive actions of the reactive component executes synchronously with the send and receive actions of the intolerant application; and the processing of negative acknowledgements is executed asynchronously with the intolerant application.

The developer of a fault-tolerance component specifies the methods that are executed synchronously and the methods that are executed asynchronously. The developer of an intolerant application provides guidelines as to what methods of an intolerant application can be executed synchronously with a fault-tolerance component. The developer of the intolerant application may specify individual methods that may be considered for synchronous composition or allow all methods of a certain type (e.g. methods from some class or all public methods) to be synchronized.

Our framework helps in the composition of a fault-tolerance component with an intolerant application. This composition can be performed statically or dynamically. Further, for a distributed application, since the fault-tolerance component is often distributed, the composition involves composing the fault-tolerance component fractions with the processes of the intolerant application. Towards this end, we instantiate a *framework node* at each process in the fault-intolerant application (cf. Figure 2.1). Each framework node consists of a *component manager*, an *adaptation module* and a *reset module* (cf. Figure 2.2). The component manager performs the addition and the removal of fault-tolerance component fractions. The adaptation module selects the fault-tolerance component based on an *adaptation strategy.* The reset module ensures that the addition and removal of fault-tolerance component is atomic, continuous and synchronized. The framework interacts with the intolerant application and the component library. The component library contains detectors and correctors along with their *contracts.* A component can be added to or removed from the library. In the following sections, we discuss the individual modules of the component framework and present their functionalities.

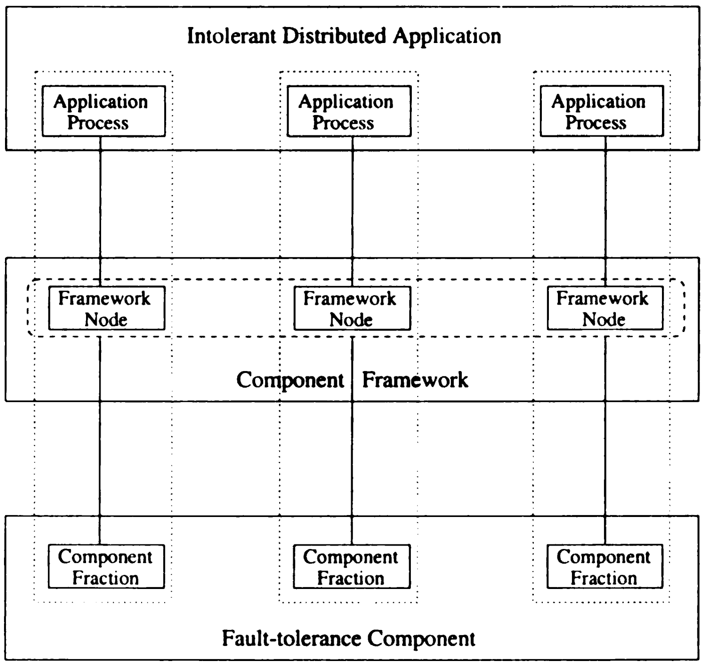

Figure 2.1: A Distributed Application Composed With Our Framework

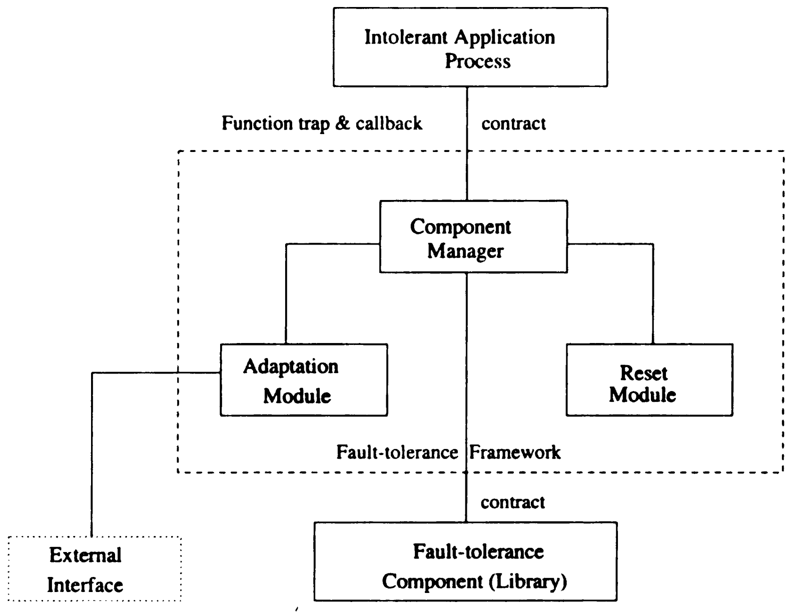

Figure 2.2: Architecture of the Framework Node

2.1 Component Manager

The function of the *component manager* is to add and remove fault-tolerance component fractions. Based on the contract with the intolerant application, it intercepts the function call of the intolerant application. Then, it calls the corresponding function of the fault-tolerance component. Subsequently, based on the application requirements, the fault-tolerance component function may call another function and/or it may call the intolerant application function that was intercepted. To facilitate dynamic composition, the component manager also communicates with other modules of the framework, namely, the *adaptation module* (Section 2.3) and the *reset module* (Section 2.4).

2.2 Contracts

The *contract* defines the constraints that are imposed on the process of a fault-intolerant application and the component fraction installed at that process. The contracts used at different processes need not be identical, i.e., it is possible that the contract between one process and its component fraction may be different from the contract between another process and its component fraction. The contract between the fault-intolerant application and the fault-tolerance component is the collection (conjunction) of individual contracts between the processes in the application and the fractions of the component.

The contract consists of two parts, an enforceable contract and an unenforceable contract. In its enforceable contract with the framework, the fault-tolerance component specifies its requirements from the fault-intolerant application. These requirements include the parameters of the expected method, their types, values and the relation between them. The contract between the fault-intolerant application and the framework specifies the methods (including their parameters) that are exposed (e.g. are public) by the fault-intolerant application. The component manager at each process verifies these enforceable contracts when it integrates the fault-intolerant application with the fault-tolerance components. The unenforceable contracts explicitly identifies the conditions under which the fault-tolerance guarantees can be provided. While these unenforceable contracts are not currently used during composition, we expect that in future version of the framework, some of the provisions in this part will be verified. Unless otherwise specified, in the rest of the paper, we use the term contract to mean the enforceable part of the contract.

2.3 Adaptation Module

The *adaptation module* is responsible for adaptation of fault-tolerance components. As noted earlier, we can have multiple fault-tolerance components that may be used to provide fault-tolerance. Moreover, the performance of a fault-tolerance component may vary depending on the environment conditions. Hence, for a given environment, we may have one fault-tolerance component that performs better than the others. The adaptation module is responsible for selecting an appropriate fault-tolerance component from the component library. Towards this end, the adaptation module needs to know the *adaptation strategy* that is specified by the user. In the current implementation, we do not provide adaptation strategies. Instead, the adaptation module provides a user interface through which the user can choose the fault-tolerance component to be added or removed. Further, for simplicity, in our current implementation we allow the adaptation module at one process to initiate the component change. It is possible to extend our framework so that any process can initiate the component change based on the approach presented in Chapter 6.

In future, we plan to provide a *learning module* alongside the adaptation module. This learning module will learn about the new components that are added to the library and will identify the adaptation strategy based on some information provided by the components in the library. It will also monitor environment changes to determine situations under which a component addition/removal will be done.

2.4 Reset Module

The reset module deals with the case where a fault-tolerance component being added or removed is distributed across the processes of a distributed application. It ensures that the addition and removal of this fault-tolerance component is atomic, continuous and synchronized. In this context, a reset operation is an operation in which an application process removes an old component and starts using a new component. The reset is initiated when the component manager has verified the contract of the new fault-tolerance component that the adaptation module has chosen. The reset module communicates with the component manager to get information, such as name of the new component and its location. The three main functions of the reset module are: (1) when the reset is in progress, the processes involved in the reset communicate only if they are using the fractions of same component; (2) upon successful completion of the reset, all processes involved use the new component, and (3) upon unsuccessful completion, all processes involved use the old component. The reset module uses the *distributed reset protocol* [3,4] to perform the above functions. We discuss this module and how it achieves the addition/removal of distributed components in Chapter 3.

2.5 Component Library

The component library acts as a repository of reusable detectors and correctors. The framework, when necessary, can select one or more of these components for static/dynamic addition. Components can be dynamically added to the library. The library also provides a *default component.* When no component is added, this default component is used. When the component manager traps an exposed method, if the default component is used, it simply calls that method back. By providing this component, we can treat addition/removal of a fault-tolerance component as a special case of component replacement.

CHAPTER 3

Reset-based Composition of Distributed Components

As mentioned earlier, a fault-tolerance component in a distributed application may itself be distributed. This introduces new challenges in the integration of these fault-tolerance components with a fault-intolerant application. While the fault-tolerance components are added or removed across processes of a distributed application, we need to ensure that the application continues to provide its functionality. We first discuss the three important challenges, namely, atomicity, minimal blocking and synchronization that arise during dynamic addition of distributed components. These challenges also apply during removal and replacement of distributed components.

Atomicity. When a distributed fault-tolerance component is added to an intolerant application, we need to ensure *atomicity* of such integration, i.e., all fractions of the distributed component should be installed across the processes of an intolerant application involved in providing fault-tolerance. In other words, if an initiating process adds its component fraction, then all other processes involved in providing fault-tolerance should add their component fraction as well. Only when all the processes have added their component fractions, can we ensure that the added component will provide the required fault-tolerance. In some cases, fault-tolerance could be provided even if some processes (including the component fraction at these processes) fail. However, in such cases, it is typically necessary to add the component fraction for each non-failed process. Hence during this discussion, we ignore this special case and assume that the component fraction must be added to each process involved in providing fault-tolerance.

Minimal Blocking. Another challenge is that during the addition of a distributed component, the application should not be blocked. While it is desirable that the addition of a distributed component be entirely non-blocking, it is not always possible to do so due to dependency among component fractions. We, therefore, require that the blocking introduced during the addition be minimal.

Synchronization. All processes involved in addition of fault-tolerance components may not add the component fraction at the same time. Hence, we will have a situation where some processes have added their component fractions, while some others have yet to add their fractions. If a process that has added a component fraction interacts with another that has not, then the results can be unpredictable. Therefore, the framework must ensure that interactions do not cross a *composition-boundary*, i.e., a process that has added a component fraction does not interact with another process that has not added the corresponding component fraction.

Our component framework uses the distributed reset protocol [3, 4] to deal with atomicity, minimal blocking and synchronization. In Section 3.1, we discuss the features of the reset protocol that are applicable in this context. In Section 3.2, we discuss the challenges imposed by distributed components and provide solutions to deal with these challenges. Then, in Section 3.3, we describe, how our framework uses the reset protocol. In Section 3.3, we also explain how we modified the reset protocol. In Section 3.4, we discuss how to deal with faults that occur during component additions, removal or replacement.

3.1 Distributed Reset Protocol

The reset-subsystem in [3, 4] can be embedded in an arbitrary distributed system in order to allow the processes to reset the system to a given global state. In the model in [3, 4], each process consists of an application module and a reset module. The application module at any process may begin the reset operation to reset the system to a given global state. The function of a reset module is to (1) reset the state of the application to a state that is reachable from a given global state, and (2) inform the application module when the reset operation is complete.

The reset solutions in [3,4] deal with atomicity, minimal blocking and synchronization. Regarding completion, each reset-operation satisfies the following two properties. (1) Every reset operation is *non-premature*, i.e., if the reset operation completes, then all processes have been reset and the program state is reachable from the given global state. (2) Every reset operation *eventually completes*, i.e., if an application module at a process initiates a reset operation, eventually the reset module at that process informs the application module that the reset operation is complete. Regarding continuous operation, the reset protocol enables the distributed application to continue functioning while the processes are individually reset. More specifically, these solutions allow the program computation to proceed concurrently with the reset, to any extent that does not interfere with the correctness of the reset. Finally, regarding synchronization, the solutions in [3,4] identify a *reset boundary*, which determines the application modules that can interact safely among themselves.

To simplify the reset operation, the reset module maintains a rooted spanning tree [5, 6] of all non-failed processes. It uses this spanning tree to perform a *diffusing computation* [7] in which each process resets its state. The diffusing computation begins at the root of the spanning tree. The root of the tree resets the state of its local application module and initiates a reset wave that propagates along the tree towards the leaves; whenever the reset wave reaches a process, the process resets the state of its local application module and propagates the reset wave to its children.

After the reset wave reaches a leaf, it is reflected as a completion wave towards the root. A process completes the reset after all of its children have completed. The reset is complete when the root receives the completion wave from all its children.

3.2 Dealing With Dependency Among Component Fractions

We use the distributed reset protocol to add, remove, or replace a fault-tolerance component across a distributed application. As discussed earlier, a fault-tolerance component consists of multiple fractions that are distributed across an application. These component fractions may exhibit some kind of dependency that restricts us from adding or removing them arbitrarily. In this section, we describe how the *dependency relation* among fault-tolerance component fractions affects their addition and removal. Then, in the next section, we explain how we use the distributed reset protocol in adding or removing a fault-tolerance component across a distributed application.

Safe states of a component. A component fraction of a distributed component cannot be removed arbitrarily as other component fractions or the local application process may depend on it. For example, if component fraction x requires a response from y to continue then removal of y can lead to incorrect functioning, e.g., deadlock, of x. Likewise, the application process where x is installed may be dependent on x. To deal with these problems, we introduce the notion of a safe state. The safe state of a component fraction is further classified as a *global safe state* and a *local safe state.* A component fraction is in a global safe state if (1) no other component fraction depends on it, and (2) the application process where that component fraction is installed does not depend on it. Thus, if a component fraction is in a global safe state, then it is safe to remove it, as its removal will not affect any other component fractions or the application. A component fraction is in a local safe state, if the application can be blocked safely at the process where the component fraction is installed. When a component fraction is in a local safe state, other component fractions may depend on it. However, in a local safe state, the application process at the component fraction does not depend upon the current state of the component fraction. Hence, the application process can be blocked (from communicating with other processes) until the new component fraction is added at that process. We assume that periodically the component fraction at each process will be in a local safe state.

To identify local safe states and global safe states, each component fraction provides a function, *checkState*, whose return value is *safetoremove* (global safe state), *safetoblock* (local safe state) or *unsafetoremove*. The return value of the *checkState* function at the component fraction j is determined based on the current state of j and on the state information of the other component fractions received by j in the past. When a component fraction is in local safe state or global safe state, the information is propagated to other component fractions. This information can, in turn, allow those component fractions to enter local/global safe states. We explain, in Section 3.3, how the reset module uses the *checkState* function during reset.

Dependency relation for a component. Now, we discuss different types of dependency relations that exist for a distributed component and how we deal with those dependency relations. For this discussion, we say that component fraction x depends on component fraction y if there exists a state where removal of y causes incorrect functioning of x.

1. **No dependency.** This is the simplest case. In this case, there exists no dependency among component fractions and they can be removed independently.

Hence, all the component fractions will (eventually) return *safetoremove* when *checkState* is invoked.

2. **Acyclic dependency for removal.** As the name suggests, this case deals with the situation where the dependency relation among component fractions is acyclic. It follows that there is at least one component fraction such that no other component fraction depends on it. This fraction can now be removed as *checkState* for this fraction will return *safetoremove.* The removal of this component fraction will, in turn, enable the removal of other component fractions, and so on.

3. **Acyclic dependency with blocking.** Consider a case where we have two component fractions x and y that are mutually dependent. In other words, x (respectively, y) cannot be removed while y (respectively, x) is still running. Further, assume that the application at x can be blocked and the knowledge that the application at x is blocked enables the removal of y. Now, we could remove the fractions x and y as follows: block application at x, remove y and remove x. In this case, initially, when *checkState* is invoked at x (respectively, y), it will return *safetoblock* (respectively, *unsafetoremove*). Later, at some point after x is blocked, *checkState* at y will return *safetoremove* and subsequently, *checkState* will return *safetoremove* at x. More generally, after introducing the notion of blocking, if the dependency relation among component fractions becomes acyclic, then the corresponding component fractions can be removed as in case 2.

4. **Cyclic dependency.** Here the component fractions exhibit mutual dependency even after introducing blocking. Hence, they cannot be removed using

any of the three cases discussed above. Possible ways to deal with such dependency are as follows:

- It is likely that the mutual dependency among component fractions does not exist during all the time while the application is running. There may be instances during run-time, when the component fractions do not depend on each other. Hence, we can add/remove component fractions during those instances.
- Another approach could be to ignore the dependency relation. Although, this approach may fail in general, if the new component is stabilizing fault-tolerant [8] then it will eventually reach a state from where it will work correctly. This approach is presented in [9].

3.3 Using Distributed Reset For Dynamic Composition

As discussed in Chapter 2, each process of an intolerant distributed application is instantiated with a *framework node* that includes a reset module. In this section, we describe how the reset module helps in dynamic addition/removal of a fault-tolerance component. We discuss replacement of a fault-tolerance component; addition and removal being special cases of replacement. For this discussion, assume that the adaptation module at a process, say X, has decided to change the fault-tolerance component. We call X the *initiator*.

The adaptation module at X informs the component manager at X about the required change. The component manager at X generates the *magic number* for the instance of a new component. The magic number is generated by using the initiator ID, the current time at the initiator, and it is used to uniquely identify the instance of the fault-tolerance component. The component manager appends this magic number in the message header while communicating with component managers at other processes of the application. The component manager at X, now, uses the reset module for changing the fault-tolerance component.

The reset consists of two waves: a *reset-initialization wave*, and a *replacement wave*. The replacement wave consists of two sub-waves, namely, a *reset-transition wave* and a *reset-completion wave*. We first present the outline describing the reset process and then explain in detail each of the reset waves. The reset module at X initiates the reset by sending the reset-initialization wave. In the reset-initialization wave, all processes change to the *transit state* and initialize the component fraction of the new distributed fault-tolerance component. Thus, in the transit state, a process has initialized the new component fraction, although it is still using the old component fraction. After all processes have set themselves into the transit state, the reset-initialization wave completes successfully. In case any process does not set itself into transit state during the reset-initialization wave, the reset-initialization wave completes unsuccessfully. The option of unsuccessful completion is provided to deal with the case where the processes need to obtain their component fraction remotely and they fail to do so. If the reset-initialization wave completes unsuccessfully, component replacement is abandoned.

Upon successful completion of the reset-initialization wave, the reset module at the initiator starts the replacement wave. The replacement wave begins with the reset-transition wave from the initiator (root) towards leaves. Each process receiving the reset-transition wave invokes the *checkState* function of the component fraction to determine the state of the component fraction. During the reset-transition wave, the processes remove the old component fraction and add the new component fraction depending on the state information returned by the function. After a leaf process has completed the replacement of its component fraction, it sends the reset-completion wave to its parent. Further, if a non-leaf process has completed the replacement of its component fraction and it has received the reset-completion wave from all of its children, it propagates the reset-completion wave to its parent. The reset-completion wave eventually reaches the initiator X. We allow another reset to start only if the first reset is completed (successfully or unsuccessfully). For simplicity, we assume that only one process can initiate the component change. This assumption can be weakened by allowing any process to change the component by using the approach in [3] where other processes communicate their decision to the initiator. We now explain the reset waves in detail.

Reset-initialization wave. The reset module at X initializes the reset by sending the reset-initialization wave to all its neighbors. The reset module uses the *component manager protocol* to communicate with other reset modules. The component manager protocol communicates information such as the name of the component, the magic number of the component, the location of the server where components are available, etc. Each process that receives the reset-initialization wave performs the following tasks:

1. It sets its parent to the first process from which it received the reset-initialization wave.

2. It propagates the reset-initialization wave to all its neighbors except its parent.

3. If a process receives the reset-initialization wave again it informs the sender of the identity of its parent. This information is used to form a tree.

4. If the process that receives the reset-initialization wave is a leaf (i.e., no neighbor process has set its parent to this process), it initializes the new component

and sets itself into the transit state, where it is still using the old component while waiting to use the new component. If the process fails to initialize the new component (e.g. if it lacks the required resources), it sets itself into the *error state.* The process that has set itself into the transit state or error state communicates its state to its parent.

5. When a process has received the transit state message from all its children, it sets itself into transit state by initializing the new component. If it receives the error state information from any of its children or if it fails to initialize the new component fraction, it sends the error state message to its parent. Eventually, the root process receives the transit state or the error state information from its children. If it receives the error state information from any of its children, it can restart the reset-initialization wave or abandon component replacement based on the threshold value set for the number of reset-initialization waves that can be initiated. In case component change is abandoned, other processes would be informed about this so that they can return to normal state. If the root process receives transit state information from all its children, it initializes the new component and sets itself into the transit state.

Reset-transition wave. When all processes are in the transit state, i.e., at the successful completion of the reset-initialization wave, the initiator X starts the reset-transition wave. This reset wave uses the function *checkState* that each component fraction provides. When a process receives the reset-transition wave, it performs following task:

1. It propagates this wave to all its children.

2. It invokes the *checkState* function, which returns one of the three values: *safetoremove*, *safetoblock* or *unsafetoremove.*

   (a) If the function returns *safetoremove*, the process removes the old component and starts using the new component. It then sets itself into the normal state. Further, it adds the magic number for the new component instance in the message header of all the messages that it sends. All the other processes that participated in the reset are now either in the transit state or the normal state (if a process has already changed its fraction). The processes in the transit state continuously check the magic number of the messages received. If the process has not started using the new component, it buffers all the messages that contain the new magic number. Note that, once an old component is discarded, the component manager does not need to buffer messages and it can forward all messages to the new component.

   (b) If the function returns *safetoblock*, the component manager blocks the application process at that component fraction. After a component fraction receives information about other component fractions being removed or other application processes being blocked, eventually, the function *checkState* at the blocking process will return *safetoremove.* When that occurs, we follow case 2a.

   (c) If the function returns *unsafetoremove*, it is periodically invoked till it returns *safetoblock* or *safetoremove*, in which case we follow the case 2b or 2a respectively. For efficiency, we call *checkState* when the intolerant application and the fault-tolerance component synchronize.

Reset-completion wave. The transition wave is reflected towards the initiator (root) as the reset-completion wave. The leaf process sends the reset-completion wave to its parent after it removes the old component fraction and starts using the new component fraction. Any non-leaf process, which completes the component fraction replacement and receives the reset-completion wave from all its children, sends the reset-completion wave to its parent. When the initiator performs the component replacement and receives the reset-completion wave from all of its children, the component replacement is complete.

Claim. *The atomicity, minimal blocking and synchronization properties are satisfied during component replacement if the component does not exhibit cyclic dependency.*

Proof sketch. The component replacement starts with an initialization wave. If the initialization wave completes unsuccessfully, then none of the component fractions is replaced. After successful completion of the initialization wave, the reset module starts a transition wave. The component fractions are replaced during the transition wave. A completion wave is propagated towards the initiator when the replacement of the component fraction is complete. In our approach, the completion wave is deferred to handle the dependency relations that exist during the component replacement. As long as the dependency is not cyclic dependency, both the transition wave and the completion wave eventually complete ensuring that all processes replace their component fractions. Thus, component replacement is atomic.

The component replacement blocks the application only when the component fraction is being replaced. Once it is safe to remove a component fraction, it is replaced and the application is unblocked. Thus, the component replacement introduces blocking only if it is required to safely replace the component fractions. It follows that component replacement is minimally blocking.

During component replacement, old component fractions do not communicate with new component fractions; such communication would violate the dependency relation. For example, if the old component fraction at process j communicates with the new component fraction at process k, it would mean that the old component fraction at k was removed while the old component fraction at j was still depending on it. Further, if the new component fraction installed at k communicates with j that is using the old component fraction, the component manager at j buffers these messages. When the new component fraction is added at j, the buffered messages are delivered to the new component fraction. Thus, component fractions of different components do not communicate with each other. Therefore, component replacement is synchronized. □

Addition and removal of distributed component. In the above discussion, we considered the replacement of a distributed component. The addition and removal are special cases of replacement. During instantiation of our framework with the application, our framework traps the functions exposed by the developer of the application and transfers the control to the *default component* that simply calls back the trapped function. Now, for addition of a distributed component, we remove the default component and replace it with the new component. In one conservative approach, the *checkState* function can be implemented as follows: Initially, all component fractions of the default component return *safetoblock*. When all the neighbors are blocked, the *checkState* function returns *safetoremove*. For applications where there is two-way communication between neighbors, this implementation of *checkState* ensures minimal blocking. For applications where there is one-way communication between some neighbors, minimal blocking can be ensured if *checkState* at a process returns *safetoremove* without waiting for the status of processes that do not communicate with it.

Remark. In the distributed reset protocol described in [3,4], a tree module maintains a spanning tree of non-failed processes and a diffusing computation is used to propagate the reset wave across all the processes. In our modified reset protocol, we generate the spanning tree on the fly during reset-initialization wave. We can also have another module that generates and maintains the spanning tree. The initiator becomes the root of the spanning tree. In this case, reset-initialization would not need to do the additional work of creating a spanning tree. All the reset waves would use the spanning tree created by the tree module.

3.4 Dealing with Faults during Component Replacement

In this section, we discuss the extension of the approach in Section 3.3 to deal with faults that occur during addition, removal or replacement. The algorithm discussed in Section 3.3 is a variation of the intolerant version of the programs in [3,4]. By treating the algorithm in Section 3.3 as an intolerant application and using the fault-tolerance components from [3,4], we can (statically) add fault-tolerance to that program.

If we were to add the fault-tolerance component from [3], the resulting algorithm will ensure that stabilizing fault-tolerance [8] is provided to faults including process/channel failures/repairs and transients. Thus, even if these faults occur, eventually the application will recover to a state from where subsequent component replacements will be correct, i.e., atomic, minimally blocking and synchronized. If we were to add the fault-tolerance component from [4], in addition to the stabilizing fault-tolerance to these faults, the resulting algorithm will provide masking fault-tolerance to process/channel failures/repairs. Thus, if only process/channel failures/repairs occur then the component replacement will be always correct. Moreover, if more general faults such as transients occur then the algorithm will recover to a state from where subsequent component replacements will be correct. Since the algorithm in Section 3.3 can be used as an input to the framework, it follows that any fault-tolerance property that can be added to the reset program can be added to our framework.

CHAPTER 4

Application of the Framework for Message Communication

In this chapter, we illustrate how we use our framework, from Chapter 2, in the context of a message communication application. We have chosen this simple application because it allows us to demonstrate most of the features of our framework including the availability of multiple distributed fault-tolerance components, the need for dynamic composition, and dependency among component fractions. We first explain the abstract version of the intolerant application and the correctors used to provide fault-tolerance in this application. Then, we explain how the application is implemented in Java.

4.1 Abstract Version of Message Communication

We will first discuss an abstract version of the intolerant message communication application, the faults and the corrector in terms of guarded commands. Each action is represented in the following form:

$$\langle name \rangle :: \langle guard \rangle \longrightarrow \langle statement \rangle$$

The guard of an action is a boolean expression over the program variables. The statement of an action updates zero or more program variables. An action can be executed only if its guard evaluates to true. To execute an action, the statement of that action is executed atomically.

Variables. For simplicity, in this section, we assume that there is only one sender and one receiver. (Our Java implementation, discussed in Section 4.2, deals with multiple receivers.) The sender process generates a message m and sends that message to the receiver process. The application maintains the following variables for each message m:

- send_in_future (m): the sender will send message m in future.
- intransit (m): the message m is in transit from the sender to the receiver.
- received (m): the receiver has received message m.

Actions. The actions of an intolerant message communication application are as follows:

$$
\begin{array}{llcl}
\textit{sender}:: & \textit{send\_in\_future}(m) & \longrightarrow & \textit{send\_in\_future}(m) := \textit{false}; \\
 & & & \textit{intransit}(m) := \textit{true}; \\
\textit{receiver}:: & \textit{intransit}(m) & \longrightarrow & \textit{intransit}(m) := \textit{false}; \\
 & & & \textit{received}(m) := \textit{true};
\end{array}
$$

Invariant. The invariant characterizes the set of all states from where the intolerant application satisfies its specification. Initially, $\forall m :: \textit{send\_in\_future}(m)$ is true. Subsequently, if $\textit{send\_in\_future}(m)$ becomes false, then $\textit{intransit}(m)$ becomes true. If $\textit{intransit}(m)$ becomes false, then $\textit{received}(m)$ becomes true. Thus, the invariant of the intolerant application is as follows (Note that this invariant is not unique; a stronger invariant that requires exactly one of these predicates to be true is also acceptable):

$$\textit{invariant}:: \quad \forall m :: \quad \textit{send\_in\_future}(m) \ \lor \ \textit{intransit}(m) \ \lor \ \textit{received}(m);$$

Fault. The invariant of this application is violated if messages are lost. The fault action is represented as follows:

$$\textit{fault}:: \quad \textit{intransit}(m) \quad \longrightarrow \quad \textit{intransit}(m) := \textit{false};$$

Correctors. Several correctors are available to provide fault-tolerance to message loss. The correction predicate, the predicate to which the intolerant application should be restored after the occurrence of faults, of a corrector that provides required fault-tolerance is:

$\forall m :: invariant(m)$

We consider two fault-tolerance components, a *proactive* component and a *reactive* component, in this example. The former is based on the idea of *forward error correction* (FEC) whereas the latter is based on the idea of retransmission. Each of these fault-tolerance components consists of component fractions that are installed at the sender and the receiver processes of the intolerant application.

The *proactive* component sends extra $(n-k)$ parity packets for each group of k data packets. If any data packet gets lost during transmission, the receiver can generate the lost data packet if it receives at least k packets from a group that contained the lost data packet. Thus, the actions of the *proactive* component are as follows (Note that this corrector is designed for the fault where no more than $(n-k)$ packets are lost from each group.):

```
sender::    ∀n, k : n, k > 0, n > k :   for each group of k data packets
                send k data packets and (n − k) parity packets;

receiver::  ¬invariant(m)  ∧  at least k packets are received from
                group containing n packets   ⟶   received(m) = true;
```

The *reactive* component detects if $invariant(m)$ is false. Subsequently, it satisfies $invariant(m)$ by setting $send\_in\_future(m)$ to true. Thus, the action of the *reactive* component is as follows:

```
sender::    ¬invariant(m)   ⟶   send_in_future(m) := true;
```

4.2 Java Version of Message Communication

In this section, we discuss how the message communication application and the two components are implemented in Java. The sender process has a function called *send* and the receiver process has a function called *receive.* These functions are exposed by the intolerant program, i.e., the fault-tolerance component can trap these functions to provide the required fault-tolerance. The component manager at the sender process, which intercepted the send function of the intolerant application, invokes the appropriate function of the fault-tolerance component that is specified by the adaptation module.

Contracts. Now, we discuss the contracts that we have defined for this application. We classify contracts into two types: one that can be verified formally, and another that cannot be verified formally. The contract that cannot be verified formally is represented as a document. One such contract states that for every call to a send function made by a sender there should be corresponding receive call made by a receiver. The contracts that can be verified formally are represented in a *meta-application file* and a *meta-component file.* The meta-application file contains the contract between our framework and the intolerant application whereas the meta-component file contains the contract between our framework and the fault-tolerance component. There is one meta-application file for each application process and one meta-component file for each component fraction.

A meta-application file is either supplied by the developer of the intolerant application or it can be generated automatically from the intolerant application. We are currently exploring efficient ways of generating this file. The entries in the meta-application file for the intolerant application process (sender) are as follows (The entries for the receiver process are similar):

sendClass:*java.net.DatagramSocket*

sendFunction:*send*

sendNumOfArguments:1

sendArguments:*java.net.DatagramPacket*

sendPacketArgumentNumber:0

...

This file specifies the name of the methods that are exposed by the sender process and other information related to these methods. In this case, the method exposed by the sender process is **send**. The details about this method such as its class, number of arguments, etc. are stored in the meta-application file.

A meta-component file is associated with each fault-tolerance component fraction. This file contains entries similar to a meta-application file. It has some parameters that are to be instantiated with appropriate parameters from the meta-application file before installing this component and some parameters that are supplied by the component developer. The entries in the meta-component file for the fraction of the proactive component at the sender process are as follows (The entries for the fraction at the receiver process and for the reactive component are similar):

functionName:*send*

functionClass:*java.net.DatagramSocket*

functionArguments:*java.net.DatagramPacket*

componentFunction: *fec-send*

...

In this particular example, during composition the **functionName** is instantiated with *send* from the meta-application file. Similar instantiations are also performed for other entries. The component uses this file to obtain information about the functions of the intolerant application. This information is provided through appropriate instantiation of the parameters of the meta-component file from the meta-application file. This file also has information about the component that is used by the framework.

The component manager at the sender process traps the *send* function of the sender process and transfers the control to the *fec-send* function of the fault-tolerance component. Similarly, at the receiver process, the component manager traps the *receive* function of the receiver process and transfers the control to the *fec-receive* function of the fault-tolerance component. When the control is transferred from the intolerant application to the component, the component performs the tasks required by FEC and, if necessary, calls the trapped function of the intolerant application to perform the actual send/receive.

The instantiation of parameters of the meta-component file is done by manual matching of the meta-application file and the meta-component file. We save the meta-component file after it has been instantiated with the application related parameters from the meta-application file. Hence, the future use of the meta-component file during the new instantiation of the component would not require any human intervention. We are exploring heuristics that will allow us to do this matching automatically with minimal human intervention.

Dependency relations. Now, we consider the dependency relation among the component fractions of the *proactive* component. The component fraction at the sender process encodes the data packets, generating $(n-k)$ parity packets for each k data packets. The component fraction at the receiver gets k data packets by decoding the group of n packets. We notice from the functions of the component fractions that there exists acyclic dependency for removal among these component fractions.

More specifically, the component fraction at the receiver process is dependent on the component fraction at the sender process. In other words, if the component fraction at the receiver process is removed before the component fraction at the sender process, then the receiver may not be able to decode the encoded packets that it might still receive from the sender process. Also, the component fraction at the sender process cannot be removed while it has processed a partial group of packets.

Now, we discuss the tasks involved in removing the proactive component (say, to add the reactive component). As discussed in previous paragraph, dependency relation in the component fractions fall in the category *acyclic dependency for removal*. Hence, we remove the component fraction at the sender before removing the component fraction at the receiver. Moreover, at the sender, *checkState* returns *unsafetoremove* when a partial group of packets is sent. When the component fraction at the sender has sent a complete group and has not started encoding another group, the *checkState* at the sender returns *safetoremove*. Finally, the knowledge about the removal of the component fraction at the sender enables the removal of the component fraction at the receiver.

Now, let us consider the dependency relation among the component fractions of the *reactive component*. The component fraction at receiver sends a negative acknowledgement for a lost packet. The component fraction at the sender retransmits the packet for which it receives a negative acknowledgement. Clearly, these component fractions are mutually dependent. The component fraction at the receiver process cannot be removed before the component fraction at the sender process is removed. Also, the component fraction at the sender process cannot be removed before the component fraction at the receiver process, since the receiver may have already sent a negative acknowledgement before knowing about the removal of the component fraction at the sender. However, in this case the *checkState* at the sender returns *safetoblock*, as the sender process can be blocked from sending messages. The knowledge about the blocking of the component fraction at the sender enables the removal of the component fraction at the receiver. Thus, the dependency relation of the reactive component falls in the category *acyclic dependency with blocking.*

Now, we discuss the tasks involved in removing the reactive component. Based on the dependency relation, we first block the application process at the sender. While the application process at the sender is blocked, the component fraction at the sender can still handle the negative acknowledgement from the receiver and can retransmit any lost packets. Eventually, the receiver will reach a state where it has recovered all the lost packets. At this point, the component fraction at the receiver can be safely removed. Subsequently, the removal of the component fraction at the receiver will allow the safe removal of the component fraction at the sender.

4.2.1 Implementation Results

In our current implementation, we have one sender and one receiver. There are three components that we have used: default component, proactive (FEC) component and reactive (ACK-NACK) component. The component manager of our framework traps the send function of the sender process and passes the control to the default component fraction installed at the sender process. The default component fraction at the sender process calls back the send function of the sender process. Similarly, the component manager traps the receive function of the receiver process and passes control to the default component fraction installed at the receiver process, which calls back the receive function of the receiver process. We replace the default component with the proactive component and vice-versa. Further, as discussed earlier, we also replace the proactive component with reactive component and vice-versa.

The sender and receiver process runs on two different machines. The machines are Intel Pentium IV with windows 2000 operating system and are connected on a LAN. We have also implemented FEC without using our framework. The FEC parameters are (20,16) and losses are emulated by dropping packets randomly. We obtain the receiver throughputs for two cases: FEC without our framework and FEC with our framework. We note from Figure 4.1 that the throughput is not affected while using our framework.

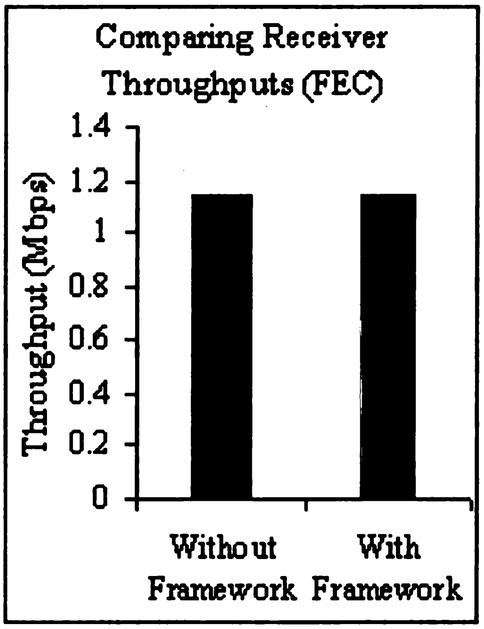

Figure 4.1: Receiver Throughput using FEC Component with/without our Framework

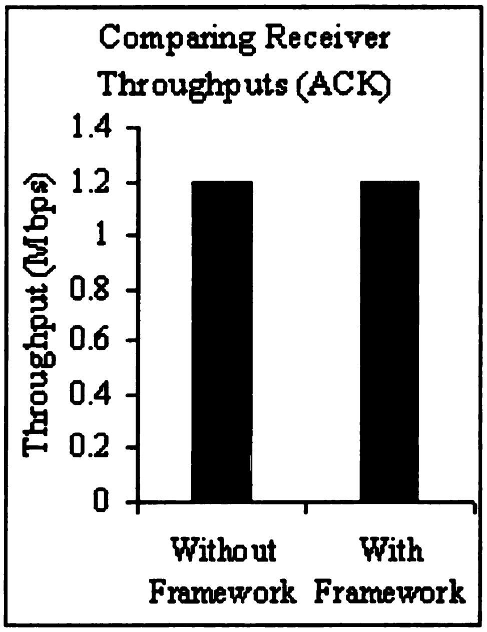

Figure 4.2: Receiver Throughput using ACK Component with/without our Framework

We obtain similar results for the reactive component (cf. Figure 4.2). Also, the delay remains unchanged while using our framework.

CHAPTER 5

Extension of the Framework for Preserving State

In Chapter 4, we described the use of our framework in the context of a message-communication application. We explained the replacement of a *proactive component* (fec-based component) with a *reactive component* (acknowledgement-based component) and vice-versa. This replacement did not require the transfer of state information from one component to another. However, in some applications we need to preserve state information of a component before replacing it with another. In this chapter, we explain how we can preserve state information during component replacement. We explain this in the context of a tree correction component.

Problems such as leader election [2], mutual exclusion [2, 10] and termination detection [2, 7] often organize processes in a tree. The application dealing with such problems has a tree component that manages this tree. There are many tree components available [11–13]. Although each of them deals with correcting a tree whenever a process fails they have some advantages and limitations over others. These tree components can be classified based on the number and types of faults they can tolerate, the time they need for recovery, and the resources they utilize. Thus, it is often beneficial to change a tree component based on the environmental conditions and application requirements.

Now, we discuss the need for preserving state while changing the tree component. If no state of the old tree component is preserved when we remove it, then information about the existing tree is lost. Therefore, after a component change, the new tree component will have to form a tree again. Until the new tree is constructed, the application may conclude that the underlying tree is broken even though no faults have occurred. This incorrect conclusion can be prevented if we extract the existing tree information from the current component and provide this information to the new component.

5.1 Issues in Preserving State During Component Change

To transfer state information from one arbitrary component to another arbitrary component is complicated, because while developing one component, one does not know about all the existing components and one cannot anticipate the new components that can be developed later. Thus, if a new component is developed then we need to perform $2N$ new transformations where N is the number of existing components. This is highly undesirable because of large number of such transformations as well as the fact that we may not have access to the details of all existing components in order to perform such transformations. To limit the situations under which the state information can be preserved, we focus on the notion of component hierarchy.

For simplicity we consider a component hierarchy of height two. At the bottom we have a *primitive component*. At the top we have multiple *enhanced components* that are derived from the primitive component. We need to perform two new transformations for each new component: conversion of a primitive component to an enhanced component and vice-versa. We impose a few restrictions on the development of components in this hierarchy. First, the primitive component has to be available during the development of the new component and its methods have to be exposed, since the new component (enhanced component) is to be derived from the primitive component. Another restriction that we impose is that the enhanced component should be able to understand the communication from the primitive component, i.e., understand messages sent by the primitive component. This is a reasonable requirement, since the enhanced component is derived from the primitive component.

We call the transformation from primitive component to enhanced component as a *scale-up* operation and the vice-versa as the *scale-down* operation. The state information of the primitive component can be easily preserved during transformation from the primitive component to the enhanced component, because of this hierarchical component development. Also, during the scale-down operation the enhanced component is able to transfer state information that can be stored by the primitive component. Intuitively, during the scale-down operation some of the properties of the enhanced component are lost.

Any tree component performs two main functions: (1) maintains parent-child relationship among processes, and (2) reconstructs the tree when any process fails (this will modify the parent-child relationship among processes). The primitive tree component in this example of tree correction stores the information about the tree. It does not do tree reformation. The enhanced tree component in addition to maintaining parent-child relationship also reconstructs a tree whenever any process fails. Moreover, a component fraction of the enhanced component can understand the communications from a component fraction of the primitive component, since the enhanced component is derived from the primitive component. However, the reverse is true only if the enhanced component communicates with the primitive component using a protocol that the primitive component can understand. In the next section, we explain how we use the notion of component hierarchy explained in this section to replace a tree component while preserving state information.

5.2 Replacing a Tree Component

We use the notion of component hierarchy discussed in Section 5.1 to replace one enhanced tree component with another enhanced tree component. We perform the following steps to replace a tree component:

1. Change the current enhanced tree component to the primitive tree component.

   - This is a scale-down operation as the current enhanced tree component is derived from the primitive tree component. At the end of this step, the primitive tree component will represent the same tree as the current tree component.

2. Change the primitive tree component to the new enhanced tree component.

   - This is a scale-up operation as the new enhanced tree component is derived from the primitive tree component. At the end of this step, the new enhanced tree component will represent the same tree as the primitive tree component.

We use the distributed reset protocol, discussed in Chapter 3, to replace the old enhanced tree component with the primitive tree component and then to replace the primitive tree component with the new enhanced tree component.

We replace a component fraction of the old enhanced tree component with a component fraction of the primitive tree component only when all of its neighbors are either blocked or are using the component fractions of the primitive tree component. We note that the dependency relation during the replacement of the old enhanced tree component with the primitive tree component can be relaxed (i.e., dependency among component fractions can be reduced), since the enhanced tree component is derived from the primitive tree component. To relax the dependency relation during replacement, the old enhanced tree component will need to communicate with the primitive tree component using the protocol that the primitive tree component can understand. In this case, we can arbitrarily remove the component fractions of the old enhanced tree component and add the primitive tree component fractions.

During the change from the primitive tree component to the new enhanced tree component, we can replace the component fractions arbitrarily because we know that the component fraction of the new enhanced tree component can interact with the component fraction of the primitive tree component. We note that, the new enhanced tree component will continue to communicate with its neighbors using the protocol that the primitive tree component can understand until it finds that all of its neighbors also use the new enhanced tree component.

5.2.1 Tree Components

In this subsection, we consider the two enhanced tree components (one is stabilizing and other is nonmasking) and develop a primitive tree component that will help us demonstrate how the notion of component hierarchy is used to preserve state information while changing from one enhanced tree component to another enhanced tree component. We first explain each of the tree components and define their *invariants* and *fault-span*, and then we describe how we change from the nonmasking to the stabilizing component and vice-versa. An invariant of a program p is a state predicate S such that $S \neq false$, S is closed in p, and every computation of p starting from a state in S satisfies the problem specification of p. Informally, an invariant of p includes the states reached in fault-free executions of p. A fault-span of program p for a fault-class F is a predicate T such that T is closed in p and F. Informally, the fault-span includes the set of states that p reaches when executed in the presence of actions in F. We consider fault actions that fail-stop and/or repair nodes, thus, change the set of up nodes:

```
Fail-stop :: up    ⟶ up := false;
Repair    :: ¬up   ⟶ up := true;
                      {initialize the state of the process}
```

For sake of simplicity, we assume that fault actions do not disconnect the set of up nodes; else, if the set is disconnected, the up nodes in each partition will reconfigure themselves into a separate rooted tree. We also assume that repair actions can reinitialize the state of the corresponding nodes.

Stabilizing tree component from [3]. In the tree correction algorithm from [3], the rooted spanning tree is represented by a "father" relation between the processes. Each *tree.i* module (fraction of the stabilizing component at the ith process) maintains a variable $f.i$ whose value denotes the index of the current father of process i. It also maintains a variable *root.i* that denotes the root of the tree it is in. The variable $d.i$ denotes length (number of hops) of the path from itself to the root. The module also maintains a variable $N.i$ that denotes the nodes that are adjacent to i. The initial graph of the father relation may be arbitrary. It is shown in [3] that starting at any state, the algorithm is guaranteed to eventually reach a state satisfying the invariant SS, where

$$\begin{aligned} SS \equiv\ & (k = max\{i | i \text{ is up}\}) \wedge \\ & (\forall i : i \text{ is up}: \\ & \quad (i = k \Rightarrow (root.i = i \wedge f.i = i \wedge d.i = 0)) \wedge \\ & \quad (i \neq k \Rightarrow (root.i = k \wedge (\exists j : j \in N.i : f.i = j \ \wedge\ d.i = d.j + 1)))) \end{aligned}$$

At each state in SS, for each process i, *root.i* equals the highest index among all up processes, *f.i* is such that some shortest path between process i and the root process *root.i* passes through the father process *f.i*, and *d.i* equals the length of this path. Therefore, a rooted spanning tree exists. Also, note that each state in SS is a fixed-point; i.e., once the *tree.i* modules reach a state in SS, no action in any of the *tree.i* modules is enabled. The fault-span of stabilizing tree component is defined as true. The spanning tree algorithm from [3] is shown in the Figure 5.1.

```
module      $tree.i(i : 1..M)$
var         $root.i, f.i : 1..M$;
            $d.i$ : integer;
            $N.i$ : list;
parameter   $j : 1..M$;

begin

    $(root.i < i) \lor$
    $(f.i = i \land (root.i \neq i \lor d.i \neq 0)) \lor$
    $(f.i \notin (N.i \cup \{i\}) \lor d.i \geq M)$          $\longrightarrow root.i, f.i, d.i := i, i, 0$
  ||
    $f.i = j \land j \in N.i \land d.i < M \land$
    $(root.i \neq root.j \lor d.i \neq d.j + 1)$          $\longrightarrow root.i, d.i := root.j, d.j + 1$
  ||
    $(root.i < root.j \land j \in N.i \land d.j < M)$     $\longrightarrow root.i, f.i, d.i := root.j, j, d.j + 1$

end
```

Figure 5.1: Stabilizing Tree Component [3].

Nonmasking tree component from [2]. The spanning tree algorithm from [2] considers an undirected, connected graph that consists of M nodes named $1, ..., M$. At each instant, each node in the graph is either "up" or "down". Two nodes in the graph are "adjacent" iff they are both up and there is an edge between them. Each node j maintains a variable $p.j$ whose value denotes the current parent of the process j. The actions of j may involve communication only with nodes that are adjacent to j. Each node maintains a variable $Adj.j$ whose value denotes the set of nodes adjacent to j. The graph of the p variables of the up nodes is a rooted tree that spans all up nodes (and the root node of the tree is its own parent).

To handle trees that are not rooted, i.e., trees that have a node whose parent is down, a variable $col.j$, for "color" of j is introduced. The value of $col.j$ is maintained by j to be green as long as all ancestors of j are up; else, the value of $col.j$ is maintained by j to be red. In other words, j colors itself red iff its parent is down or is colored red.

To handle multiple rooted trees, a variable $root.j$, for "root" of j is introduced to merge trees. The value of $root.j$ is maintained by j to be the index of the root node of the tree that j is in. The spanning tree algorithm from [2] is shown in the Figure 5.2.

The fault-span of the nonmasking tree component is:

$TN \quad = \textit{graph of the parent variables of up nodes is a forest}$

$\quad\quad \wedge\ (\forall j : j \textit{ is up} : TN1.j \wedge TN2.j \wedge TN3.j \wedge TN4.j)$, where

$TN1.j \quad = (col.j = red \ \Rightarrow (p.j \notin Adj.j \cup \{j\} \vee col.(p.j) = red))$

$TN2.j \quad = (p.j = j \quad \Rightarrow root.j = j)$

$TN3.j \quad = (p.j \neq j \quad \Rightarrow root.j > j)$

$TN4.j \quad = (p.j \in Adj.j \ \Rightarrow (root.j \leq root.(p.j) \vee col.(p.j) = red$

The invariant of the nonmasking tree component is:

$SN \quad = TN \wedge (\forall j :: SN1.j \wedge SN2.j \wedge SN3.j)$, where

$SN1.j \quad = (col.j = red \Leftarrow (p.j \notin Adj.j \cup \{j\} \vee col.(p.j) = red))$

$SN2.j \quad = col.j = green$

```
node        $j(j : 1..M)$
var         $root.j, p.j : 1..M$;
            $col.j : \{green, red\}$;
            $Adj.j$ : list;
parameter   $k : 1..M$;

begin

    $col.j = green \wedge$
    $(p.j \notin Adj.j \cup \{j\} \vee col.(p.j) = red)$    $\longrightarrow$ $col.j := red$
 ||
    $col.j = red \wedge$
    $(\forall k : k \notin Adj.j \vee p.k \neq j)$           $\longrightarrow$ $col.j, p.j, root.j := green, j, j$
 ||
    $k \in Adj.j \wedge root.j < root.k \wedge$
    $col.j = green \wedge col.k = green$                  $\longrightarrow$ $p.j, root.j := k, root.k$

end
```

Figure 5.2: Nonmasking Tree Component [2].

$SN3.j \quad = \forall k : k \in Adj.j \Rightarrow root.j = root.k)$

Primitive tree component. The primitive tree component from which the two enhanced tree components (stabilizing and nonmasking) are derived is shown in the Figure 5.3.

The invariant SP of the primitive tree component is:

$SP \equiv$ *graph of the parent variables of up nodes is a forest* $\wedge$

$(k = max\{i|i \text{ is up}\}) \wedge$

$(\forall i : i \text{ is up}:$

$(i = k \Rightarrow (r.i = i \wedge father.i = i)) \wedge$

$(i \neq k \Rightarrow (r.i = k \wedge (\exists j : j \in adjacent.i : father.i = j)))$

```
module       $tree.j(j : 1..M)$
var          $r.j, father.j : 1..M$;
             $adjacent.i$ : list;
parameter    $k : 1..M$;

begin
  NoActions
end
```

Figure 5.3: Primitive Tree Component

The fault-span of the primitive tree component is defined as true. The primitive component performs no action. It only maintains the parent-child relationship. Each fraction of the primitive component also maintains a root value that denotes root of the tree.

5.2.2 Changing from Nonmasking to Stabilizing Tree Component

In this subsection, we discuss how the nonmasking component is replaced by the stabilizing component. In the first step we change the nonmasking tree component to the primitive tree component and then in the second step we change the primitive tree component to the stabilizing tree component.

Replacing nonmasking component by primitive component. In replacing from the nonmasking component to the primitive component, we have to preserve the tree maintained by the nonmasking component. In other words, the state of the tree has to be transferred from the nonmasking component to the primitive component. During reset-transition wave (cf. Section 3.3) the `initialize` routine (cf. Figure 5.4) of the primitive component fraction is invoked. This routine copies the values of the variables ($p.j, root.j$, and $Adj.j$) from the corresponding component fraction of the nonmasking component. The reset module reads these variables from the nonmasking component fraction and passes these values as arguments to the **`initialize`** routine of the primitive component fraction.

When the **`initialize`** routine of the primitive component fraction at the leaf process is complete, the leaf process sends the reset-completion wave to its parent. When a process receives reset-completion wave from all its children, and the **`initialize`** routine of the primitive component fraction at that process is complete, then that process propagates the reset-completion wave to its parent. Eventually, the initiator receives the reset-completion wave from all its children. When the **`initialize`** routine of the primitive component fraction at the initiator is complete and it has received reset-completion wave from all its children, the replacement from the nonmasking component to the primitive component is complete. At this point, the initiator starts the step 2 to replace the primitive component with the stabilizing component.

Claim 1. *If the invariant of the nonmasking component is satisfied before the change and no faults occur during change then the invariant of the primitive component is satisfied after the change.*

Proof. If the invariant of the nonmasking component is satisfied and no faults occur during the change, then no actions of the nonmasking component are enabled and the nonmasking component represents a rooted spanning tree. The execution of the actions of the **`initialize`** routine of the primitive component fractions maps the original tree represented by the nonmasking component to a tree represented by the primitive component. If a node was a root in the nonmasking component, it stays as a root in the primitive component. Also, the parent-child relationship among nodes remains the same. Thus, if we begin in a state where SN is satisfied and no faults occur during the change, then SP will be satisfied after the change. □

Claim 2. *If the invariant of the nonmasking component is not satisfied before the change or faults occur during the change then the fault-span of the primitive component is satisfied after the change.*

Proof. Since the fault-span of the primitive component is defined as true, the claim is trivial. □

```
module      $initialize.j(j : 1..M)$
var         $r.j, father.j : 1..M$;
            $adjacent.i$ : list;
parameter   $k : 1..M$;
input       $p.j, root.j : 1..M$;
            $Adj.j$ : list;

begin
  $true \longrightarrow father.j, r.j, adjacent.j := p.j, root.j, Adj.j$
end
```

Figure 5.4: Initializing Primitive Component From Nonmasking Component

Replacing primitive component by stabilizing component. During the replacement from the primitive component to the stabilizing component, the values of the variables of the primitive component are copied into the variables of the stabilizing component as follows:

$f.i = father.i; root.i = r.i; N.i = adjacent.i;$

To initialize the variables, each component fraction of the stabilizing component has the `initialize` routine (cf. Figure 5.5). This routine is called during reset-transition wave. While initializing the variables, the following two properties has to be ensured:

- If the invariant of the primitive component is satisfied and no faults occur during the change, then the actions of the `initialize` routine should preserve the invariant of the stabilizing component.

- If the invariant of the primitive component is not satisfied or if faults occur during the change, then the actions of the `initialize` routine should preserve the fault-span of the stabilizing component.

Now, to check whether the invariant of the primitive component is satisfied, we introduce two new routines **get_snapshot** and **check_invariant** in all fractions of the primitive component. During the reset-initialization wave, the reset module will collect the snapshot of all the component fractions. After the reset-initialization wave is completed, the initiator invokes the **check_invariant** routine of the component fraction installed at the initiator. The snapshot of all the processes collected during reset-initialization is passed to the **check_invariant** routine, which checks if the invariant of the primitive component is satisfied or not. The return value of the **check_invariant** is either true or false. (To check if the invariant is satisfied, we need **check_invariant** routine only at the initiator. However, since the initiator is not fixed, we provide **check_invariant** routine at all processes.) The value returned by **check_invariant** is passed to the `initialize` routine of all fractions during reset-transition wave. Further if the invariant of the primitive component is not satisfied (i.e., either the tree is broken or their are cycles) or faults occur during the change, then the stabilizing component should preserve its fault-span, namely true.

```
module      $initialize.i(i : 1..M)$
var         $root.i, f.i : 1..M$;
            $d.i$ : integer;
            $N.i$ : list;
parameter   $k : 1..M$;
input       $father.i, r.i : 1..M$;
            $adjacent.i$ : list;
            $prim\_invariant : true, false$
begin

  $SI1$ : $prim\_invariant = true \wedge$
        $father.i = i \wedge r.i = i$          $\longrightarrow f.i, root.i, N.i, d.i :=$
                                                 $father.i, r.i, adjacent.i, 0$
$\|$
  $SI2$ : $prim\_invariant = true \wedge$
        $father.i \in adjacent.i \wedge r.i = root.(father.i)$
                                             $\longrightarrow f.i, root.i, N.i, d.i :=$
                                                 $father.i, r.i, adjacent.i, d.(father.i) + 1$
$\|$
  $SI3$ : $prim\_invariant = false \vee$
        $(father.j \notin adjacent.j \cup \{j\}) \vee$
        $(father.j = j \wedge r.j \neq j) \vee$
        $(father.j \neq j \wedge j \geq r.j) \vee$
        $(father.j \in adjacent.j \wedge r.j > root.(p.j))$
                                             $\longrightarrow f.j, root.j, N.j, d.j := j, j, adjacent.j, 0$

end
```

Figure 5.5: Initializing Stabilizing Component from Primitive Component

The `initialize` routine of the stabilizing component is shown in the Figure 5.5. Actions $SI1$ and $SI2$ are executed when the original tree is good (i.e., original tree is a rooted spanning tree). Actions $SI3$ is executed when the original tree is not good or when faults have occurred during the change. The `initialize` routine returns true if any of the actions is executed. If no actions are executed, then the `initialize` routine returns false. The `initialize` routine is called repeatedly till it returns true.

These actions of the `initialize` routine are executed only when their corresponding guard is true. For action $SI2$, its guard remains false till the *father.i* variable of the father process is set and hence the action $SI2$ will not be executed. If the invariant of the primitive component is true then the sequence of events will be as follows: First the root process will execute the action $SI1$. Then all the children of the root process will execute the action $SI2$. This will propagate all the way down to the leaf processes.

When the leaf process has completed its initialization (i.e., its `initialize` routine returns true), it sends reset-completion wave to its parent. When a process receives reset-completion from all its children and it has completed its initialization, it propagates the reset-completion wave to its parent. Eventually, the initiator will receive reset-completion wave from all its children and will complete its initialization. At this point, the component replacement is complete.

Remark. We note here that the fault-span of the stabilizing component is *true*. Therefore, we can arbitrarily initialize the variables of the stabilizing component and still stay in the fault-span.

Claim 3. *If the invariant of the primitive component is satisfied before the change and no faults occur during the change then the invariant of the stabilizing component is satisfied after the change.*

Proof. If the invariant of the primitive component is satisfied and no faults occur during the change, then from Claim 1, the primitive component represents the original tree represented by the nonmasking component. The primitive component executes no actions.

The action $SI1$ of the `initialize` routine of the stabilizing component is enabled at the root process. The execution of the action $SI1$ maps the root of the tree represented by the primitive component to the root of the tree represented by the stabilizing component. The action $SI1$ establishes $i = k \Rightarrow (root.i = i \land f.i = i \land d.i = 0)$, which is in the invariant SS of the stabilizing component.

The non-root processes have none of the actions enabled until the root executes the action $SI1$. After the root executes the action $SI1$, children of the root (i.e., fractions at a distance (in a tree) of 1 from the root) have the action $SI2$ enabled. The execution of the action $SI2$ maps the children of the root of the tree represented by the primitive component to the children of the root of the tree represented by the stabilizing component. The action $SI2$ establishes $\forall i : i \textit{ is up and } i \textit{ is at a distance of 1 from the root} : (i \neq k \Rightarrow (root.i = k \land (\exists j : j \in N.i : f.i = j \land d.i = d.j + 1))$, which is in the invariant of the stabilizing component.

After the children of the root executes the action $SI2$, their children (i.e., fractions at a distance of 2 from the root) have the action $SI2$ enabled. Subsequently, the execution of the action $SI2$ at fractions at a distance of 2 from the root enables the action $SI2$ at fractions at a distance of 3 from the root. Eventually, the execution of the action $SI2$ at the non-root processes maps the non-root nodes of the tree represented by the primitive component to the non-root nodes of the tree represented by the stabilizing component. Thus, if a node was a root in the primitive component, it stays as a root in the stabilizing component. Also, the parent-child relationship among nodes remains the same. Further, as the invariant of the primitive component is satisfied and no faults occur during the change, the only actions of the stabilizing component that are enabled and executed are $SI1$ and $SI2$. When all fractions have completed their `initialize` routine (i.e., they have executed either the action $SI1$ or $SI2$) the invariant of the stabilizing component is established.

Thus, if we begin in a state where SP is satisfied and no faults occur during the change, then SS will be satisfied after the change. □

Claim 4. *If the invariant of the primitive component is not satisfied before the change or faults occur during the change then the fault-span of the stabilizing component is satisfied after the change.*

Proof. The primitive component executes no actions. If the invariant of the primitive component is not satisfied or faults occur during the change, then the action $SI3$ of the stabilizing component is executed that establishes: $root.i = i \wedge f.i = i \wedge d.i = 0$, which is in the fault-span, namely true. □

5.2.3 Changing from Stabilizing to Nonmasking Component

In this subsection, we discuss replacement of the stabilizing component with the nonmasking component. As discussed in Section 5.2.2, this is a two step process. We first change from the stabilizing component to the primitive component and then we replace the primitive component by the nonmasking component.

Replacing stabilizing component by primitive component. In replacing from the stabilizing component to the primitive component, the state of the tree has to be transferred from the stabilizing component to the primitive component. During reset-transition wave, the `initialize` routine (cf. Figure 5.6) of the primitive component fraction is invoked. This routine copies the values of the variables ($f.j, root.j$, and $N.j$) from the corresponding component fraction of the stabilizing component. The reset module reads these variables from the stabilizing component fraction and passes the values as arguments to the `initialize` routine of the primitive component fraction.

```
module      initialize.j(j : 1..M)
var         r.j, father.j : 1..M;
            adjacent.i : list;
parameter   k : 1..M;
input       f.j, root.j : 1..M;
            N.i : list;

begin
   true ⟶ father.j, r.j, adjacent.j := f.j, root.j, N.j
end
```

Figure 5.6: Initializing Primitive Tree Component From Stabilizing Component

As the fractions of the primitive component complete their `initialize` routine, the reset-completion wave (as discussed in Section 5.2.2) is propagated towards the initiator. When the `initialize` routine of the primitive component fraction at the initiator is complete and it has received reset-completion wave from all its children, the replacement from the stabilizing component to the primitive component is complete. At this point, the initiator starts the step 2 to replace the primitive component with the nonmasking component.

Claim 5. *If the invariant of the stabilizing component is satisfied before the change and no faults occur during change then the invariant of the primitive component is satisfied after the change.*

Proof. If the invariant of the stabilizing component is satisfied and no faults occur during the change, then no actions of the stabilizing component are enabled and the stabilizing component represents a rooted spanning tree. The execution of the actions of the `initialize` routine of the primitive component fractions maps the original tree represented by the stabilizing component to a tree represented by the primitive component. If a node was a root in the stabilizing component, it stays as a root in the primitive component. Also, the parent-child relationship among nodes remains the same. Thus, if we begin in a state where SS is satisfied and no faults occur during the change, then SP will be satisfied after the change. □

Claim 6. *If the invariant of the stabilizing component is not satisfied before the change or faults occur during the change then the fault-span of the primitive component is satisfied after the change.*

Proof. Since the fault-span of the primitive component is defined as true, the claim is trivial. □

Replacing primitive component by nonmasking component. The replacement of the primitive component by the nonmasking component is similar to the replacement of the primitive component by the stabilizing component as discussed in Section 5.2.2.

The `initialize` routine of the nonmasking component is shown in the Figure 5.7. Actions $NI1$ and $NI2$ are executed if the invariant of the primitive component is satisfied and no faults occur during the change. Action $NI1$ is executed at the root process and action $NI2$ is executed at the non-root processes. If the invariant of the primitive component is not satisfied or if faults occur during the change, then the `initialize` routine of the nonmasking component has to ensure that it preserves the fault-span. In this case, the action $NI3$ is executed.

If the invariant of the primitive component is true then the sequence of events will be as follows: First the root process will execute the action $NI1$. Subsequently all the children of the root process will execute the action $NI2$. This will propagate all the way down to the leaf processes, which will then execute the action $NI2$. The leaf process sends reset-completion wave to its parent when it has completed its initialization. When a process receives reset-completion wave from all its children and it has completed its initialization, it propagates the reset-completion wave to its parent. Eventually, the initiator will receive reset-completion wave from all its children and will complete its initialization. At this point the component replacement is complete.

```
module      $initialize.j(j : 1..M)$
var         $root.j, p.j : 1..M$;
            $col.j : \{green, red\}$;
            $Adj.j$ : list;
parameter   $k : 1..M$;
input       $father.j, r.j : 1..M$;
            $adjacent.j$ : list;
            $prim\_invariant : true, false$
begin

  $NI1$ : $prim\_invariant = true\ \wedge$
          $father.j = j\ \wedge\ r.j = j$          $\longrightarrow$ $p.j, root.j, Adj.j, col.j :=$
                                                  $father.j, r.j, adjacent.j, green$
||
  $NI2$ : $prim\_invariant = true\ \wedge$
          $father.j \in adjacent.j\ \wedge$
          $col.(p.j) = green\ \wedge\ r.j = root.(p.j)$
                                               $\longrightarrow$ $p, j, root.j, Adj.j, col.j :=$
                                                  $father.j, r.j, adjacent.j, green$
||
  $NI3$ : $(prim\_invariant = false)\ \vee$
          $(father.j \notin adjacent.j \cup \{j\})\ \vee$
          $(father.j = j\ \wedge\ r.j \neq j)\ \vee$
          $(father.j \neq j\ \wedge\ j \geq r.j)\ \vee$
          $(father.j \in adjacent.j\ \wedge\ r.j > root.(p.j))$
                                               $\longrightarrow$ $p.j, root.j, Adj.j, col.j :=$
                                                  $j, j, adjacent.j, red$

end
```

Figure 5.7: Initializing Nonmasking Component from Primitive Component

Remark. We note that, unlike as discussed in Section 5.2.2 for the stabilizing component, here we cannot initialize the variables of the nonmasking component arbitrarily to preserve the fault-span.

Claim 7. *If the invariant of the primitive component is satisfied before the change and no faults occur during the change then the invariant of the nonmasking component is satisfied after the change.*

Proof. If the invariant of the primitive component is satisfied and no faults occur during the change, then from Claim 5, the primitive component represents th original tree represented by the stabilizing component. The primitive component executes no actions.

The action $NI1$ of the `initialize` routine of the nonmasking component is enabled at the root process. The execution of the action $NI1$ maps the root of the tree represented by the primitive component to the root of the tree represented by the nonmasking component. The action $NI1$ establishes $root.j = j \wedge p.j = j \wedge col.j = green$, which is in the invariant SN of the stabilizing component.

The non-root processes have none of the actions enabled until the root executes the action $NI1$. After the root executes the action $NI1$, children of the root (i.e., fractions at a distance (in a tree) of 1 from the root) have the action $NI2$ enabled. The execution of the action $NI2$ maps the children of the root of the tree represented by the primitive component to the children of the root of the tree represented by the stabilizing component. The action $NI2$ establishes $\forall j$: j *is up and* j *is at a distance of 1 from the root* : $(p.j \in Adj.j \wedge root.j = root(p.j) \wedge col.j = green)$, which is in the invariant SN of the stabilizing component.

After the children of the root executes the action $NI2$, their children (i.e., fractions at a distance of 2 from the root) have the action $NI2$ enabled. Subsequently, the execution of the action $NI2$ at fractions at a distance of 2 from the root enables the action $NI2$ at fractions at a distance of 3 from the root. Eventually, the execution of the action $NI2$ at the non-root processes maps the non-root nodes of the tree represented by the primitive component to the non-root nodes of the tree represented by the nonmasking component. Thus, if a node was a root in the primitive component, it stays as a root in the nonmasking component. Also, the parent-child relationship among nodes remains the same. Further, as the invariant of the primitive component is satisfied and no faults occur during the change, the only actions of the nonmasking component that are enabled and executed are $NI1$ and $NI2$. When all fractions have completed their `initialize` routine (i.e., they have executed either the action $NI1$ or $NI2$) the invariant of the nonmasking component is established.

Thus, if we begin in a state where SP is satisfied and no faults occur during the change, then SN will be satisfied after the change. $\square$

Claim 8. *If the invariant of the primitive component is not satisfied before the change or faults occur during the change then the fault-span of the nonmasking component is satisfied after the change.*

Proof. The primitive component executes no actions. If the invariant of the primitive component is not satisfied then the actions $NI1$ and $NI2$ are not enabled at any processes. Further if the invariant of the primitive component is not satisfied or faults occur during the change, the action $NI3$ of the nonmasking component is enabled and executed. The execution of action violates the invariant of the nonmasking component. However, the action $NI3$ establishes $p.j = j \land root.j = j \land col.j = red$, which is in the fault-span TN of the nonmasking component. When all fractions have completed their `initialize` routine, the fault-span TN is established. Thus, if we begin in a state where SP is not satisfied or faults occur during the change, then TN will be satisfied after the change. $\square$

Thus, by using the notion of component hierarchy, we can transfer the state information from one component to another. We note that during replacement:

- If the invariant of the original component was satisfied, i.e., if the tree was good before a change, and no faults occur during the change, then the invariant of the new component is satisfied, i.e., the tree remains good after the change.

- If the invariant of the original component was not satisfied, i.e., if the tree was broken before a change, or if faults occur during the change, then the invariant of the new component will eventually be satisfied, i.e., the tree will eventually be corrected due to the actions of the new component.

In this chapter, for simplicity we considered the component hierarchy of two components. However, we can easily extend it to more than two components. In this case, if two components $X1$ and $X2$ are derived from a component X in a component hierarchy, then during the change from $X1$ to $X2$, we can do the scale-down operation from $X1$ to X and then the scale-up operation from X to $X2$.

CHAPTER 6

Discussion

The framework proposed in this paper raises several questions about how it can be used and modified to suit different applications. We discuss some of these questions below.

Who initiates the component change? Can any process initiate the component change?

Although in Section 3.3, we assumed that only one process initiates the component change, it is possible to extend it so that other processes can also initiate a component change. Towards this end, we use the approach in [3] where the processes are arranged in a tree. In this case, any process that wants to change a component sends its request to the root. The root process then initiates the component change as mentioned in Section 3.3. This approach also takes care of network partitioning where each partition has its own root process that can perform the component change for that partition.

Is the reset-initialization wave necessary? What are its advantages and disadvantages?

As discussed in Section 3.3, a reset-initialization wave is used to initialize the components and create a spanning tree. We note that, the reset-initialization wave is not a requirement for our framework. If we assume that all processes already have the component fractions initialized, then we do not need the reset-initialization wave. If the components are changed frequently and all processes can always succeed in installing the new component, then it would be more efficient to remove the reset-initialization wave. However, if we remove the reset-initialization wave then the overhead incurred by the component manager will increase as it has to check the incoming messages in all cases to determine if a component change is in progress. Also in this case, the process will not have an option to abort the component change. Thus, the decision of using a reset-initialization wave is a tradeoff in performance and flexibility rather than a requirement.

Can dynamic composition be improved if the components being added are backward compatible with the current component?

Yes. If the new component is backward compatible, then we can add the new component without dealing with the dependency relation among the component fractions. This is due to the fact that the new component fraction at a process can interact with the current component fractions at other processes. Note however that, in general backward compatibility is not satisfied. Hence, in many situations, the component fractions of the new component cannot interact with the component fractions of the current component. The framework can be enhanced to simplify composition in this special case. We have not considered this issue in this paper since we are mainly interested in providing dynamic composition in cases where the new component and the current component are not related.

CHAPTER 7

Related Work

In this chapter, we discuss work related to the framework discussed in this paper. We also point out how our framework differs from the previous work.

7.1 Related work on Composition of Fault-tolerance

FRIENDS [14] (Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications) is a software-based architecture for implementing fault-tolerant applications. The software-based architecture of [14] is composed of subsystems and libraries of metaobjects. Common services required for implementing the metaobjects are provided by the subsystems. A library of fault-tolerance strategies consisting of metaobject classes is implemented on top of the corresponding subsystem. For this reason, the programmer developing a library needs to be aware of the underlying subsystem implementation. The system layer in [14] consists of three necessary sub-systems, namely, fault tolerant sub-system, secure communication subsystem and group-based distribution sub-system. The user layer is divided into application layer and metaobject layer.

In [15], Panwar, Agha and Sturman, describe a language framework, MAUD (Meta-level Architecture for Ultra Dependability), for dependable systems by focusing on modularity and composition. In [15], the meta-level of the system consists of aspects that are relevant to fault-tolerance. Thus, in [15], the base objects specify application specific functionality whereas the meta-level objects specify the fault-tolerance protocols. In their prototype implementation of MAUD, an application and its dependability protocols are linked together at compile time.

Our framework differs from the approaches in [14,15] in that our framework deals with the addition/removal of distributed components by considering the dependency relation among component fractions in a distributed component. Also, in our framework, the intolerant application and the fault-tolerance component specify a contract that describes what the latter needs from the former and what it can use while providing fault-tolerance. Based on the need for separation of concerns, the contract of the intolerant application is developed by the developer of the intolerant application and the contract of the fault-tolerance component is developed by the developer of the fault-tolerance component.

7.2 Adaptive Programming

Gouda and Herman [9] consider the problem of adding/removing distributed stabilizing fault-tolerance components. By definition, starting from an arbitrary state, a stabilizing component recovers to a state from where its subsequent computation satisfies its specification. Thus, even if one ignores the dependency relation among component fractions, after the addition of a new stabilizing fault-tolerance component, it will recover to a legitimate state. It follows that even if the dependency relation among component fractions is cyclic and the components being added are stabilizing fault-tolerant, then the techniques in [9] can be used to dynamically add/remove those components.

In their approach, some incorrect computation can occur during a component change, because they ignore the dependency relation among component fractions. In contrast, by taking the dependency relation among component fractions into consideration, we ensure that the change of a component does not cause an incorrect operation. Moreover, our framework can also deal with components that are not stabilizing fault-tolerant.

7.3 Related Work on Adaptation in Distributed Systems

Chen et al. [16] have presented an adaptation process that consists of change detection, agreement, and adaptive action. Our approach of changing components is orthogonal to the approach in [16]. In [16], either the dependency relation is ignored or is handled implicitly. This can lead to excessive blocking or incorrect results during component change. We explicitly account for any dependency during component change while ensuring minimal blocking. Secondly, in their approach, faults that occur during the change are not considered. Our approach deals with faults that can occur during component change (cf. Section 3.4). Unlike in [16], the reset module described in the paper deals only with the adaptive action that replaces the fault-tolerance component. However, the approach in [16] for change detection and agreement can be combined with our work to build adaptive component-based distributed systems. Further, our work on component change can be used in [16] to ensure that the dependency relation is correctly handled.

7.4 Electronic Switching Systems

Examples such as ESS (Electronic Switching Systems) [17] support dynamic addition/removal of components. However, in these examples, the system consists of a set of applications. When a new component is added, old applications continue to run using the old component whereas the new applications will use the new component.

When all the old applications terminate, the old component can be removed. Thus, we can view this system as a set of two disjoint systems; one using the old component and the other using the new component. By contrast, in our framework, there is only one (long-running) application that needs to change the component dynamically. Hence, we cannot use a solution where the old and new components execute concurrently until the applications using the old component terminate.

7.5 Composition at Binary Level

Modifying the source code to allow composition is a difficult task, because firstly the source code is rarely available and secondly because of cryptic nature of source code. Although the composition of components at the binary level is also not any easy task, it is highly desirable and the binary code is easily available. Binary component adaptation (BCA) [18] allows components to be composed in binary form. BCA rewrites component binaries before (or while) they are loaded and it does not require any source code access. Keller and Hölzle [18] describe the Java implementation of BCA and explain how BCA can improve the reusability of Java components. Rather than creating new classes such as wrapper classes, the definition of the original class is modified for allowing component adaptation. In the implementation of BCA, a class file is modified before it it passed on to the native JDK loader (cf Figure 7.1). The modifier operates on the internal representation of the class that the loader builds. The class loader parses the Java class file and stores the various components in an object hierarchy. Each component of the class file format, such as fields, methods, attributes, and various other entries, is represented by a C++ class.

We have studied some of the issues to allow binary level composition of distributed components using our framework. In future work, we will implement the composition

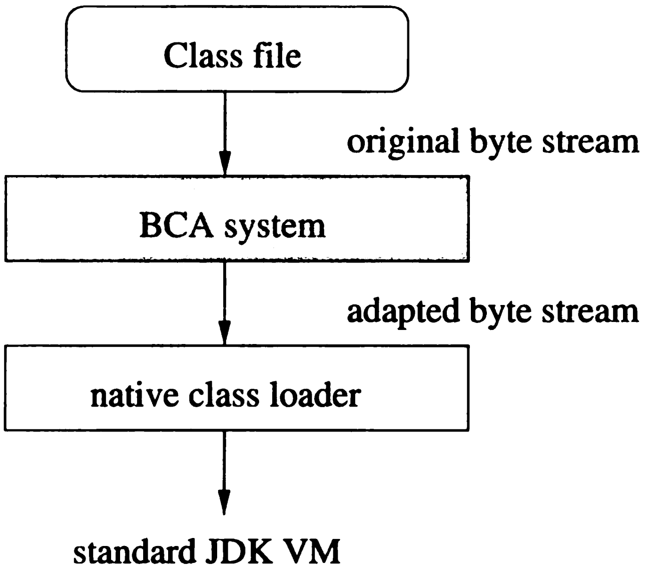

Figure 7.1: Integration of BCA system into JDK VM [18]

at binary level. We will use techniques discussed in [18] to make changes to the binary so that we can integrate our framework with an intolerant application and apply similar techniques to the component binaries.

7.6 Reflection and Run-time Composition in Java

We used *computational reflection* [19] to implement our framework. A reflective system is a system that incorporates structures representing itself. Self-representation makes it possible for the system to reason about and act upon itself. A reflective system is causally connected to the underlying behavior it describes. Causal connection implies that changes made to the self-representation are immediately mirrored in the underlying system's actual state and behavior, and vice-versa.

Reflection is one of the features in Java programming language. It allows an executing Java program to examine or "introspect" itself, and manipulate its internal properties. We use the reflection feature of Java to dynamically add Java components. Additional information about Java reflection is available in [20, 21].

7.6.1 Byte-code Rewriting

In this section, we discuss how Kava [22] uses byte-code rewriting to add behavioral reflection to Java. Load-time byte-code rewriting techniques can be used to adapt and customize the behavior of Java classes as discussed in [22]. One advantage of this approach is that it doesn't require a modified compiler or JVM. Kava is implemented on top of a standard JVM.

The Kava system allows each object or class to be bound to a metaobject. At the meta level runtime behaviors such as method invocation, method execution, field access, etc. can be redefined by the metaobject implementation. The metaobject implementation is constructed using reified aspects of the runtime object model.

Kava implements behavioral reflection in Java using byte code transformation as the underlying technique. In future work, we will use techniques discussed in [22] to make behavioral changes to the intolerant application and to the binary distributed components to allow them to be composed dynamically with our framework.

CHAPTER 8

Conclusion and Future Work

This thesis presented an adaptive component-based framework that can dynamically add, remove, and replace distributed components to an application. The framework ensured that changing a component satisfies the three properties, atomicity, minimal blocking and synchronization. The framework also ensured that the dependency relation among component fractions of a given component is correctly handled during dynamic composition. Moreover, as discussed in Section 3.4, the framework can be tailored to deal with faults that occur during a component change.

The thesis illustrated this framework in the context of fault-tolerant applications where a fault-intolerant application is composed with fault-tolerance components. The examples of message communication (cf. Chapter 4) was used to illustrate the availability of multiple components, need for dynamic composition, and different dependency relations among component fractions. The thesis also illustrates the use of the framework in Siesta, Simple NEST Application Simulator [23], developed at Vanderbilt University. In this application, the system consists of 50 nodes. Fault-tolerance routing components are developed for Siesta (cf. Appendix B) and it is shown how to dynamically change these components using the approach presented in this thesis. Although in these examples, only one fault-tolerance component was used at a time, the framework does permit the case where multiple fault-tolerance components are used simultaneously. For example, in the context of message communication, the thesis hierarchically composed the *proactive* component and the *reactive* component, discussed in Chapter 4. The thesis also proposes the extension of the framework using the notion of component hierarchy to preserve state information.

In the context of fault-tolerant applications, the framework enables independent development of the corresponding fault-intolerant application and the corresponding fault-tolerance components. Towards this end, the designer of a fault-intolerant application provides a set of methods that can be used while adding fault-tolerance components. Also, the fault-tolerance component specifies the set of methods it expects from the fault-intolerant application. During composition, this information is matched —for method renaming, parameter renaming, etc— to determine how the fault-intolerant application and the fault-tolerance component are composed. Within these constraints, the development of the fault-intolerant application and the fault-tolerance components can proceed independently.

The framework also enables the reuse of fault-tolerance components. The components used for message communication are clearly applicable in several other domains. A component, similar to that presented in Appendix B, is used for routing in ad-hoc networks [24].

There are several possible extensions to this work. Currently, the source code is preprocessed so that the functions exposed by the fault-intolerant application are trapped and each process is composed with the fraction of the default component (cf. Chapter 2). To increase the applicability of the framework, the use of binary version of the fault-intolerant application is being considered. Specifically, the approaches such as that in [18, 22, 25] are used so that a given binary is modified in such a way that the exposed functions are appropriately trapped.

While this framework is written in Java, it has the potential to be applicable in adding fault-tolerance to a fault-intolerant application in languages such as C/C++.

However, as the current approach uses Java reflection while composing components, new approaches are being devised that will allow the composition of components in these languages.

Although, the discussion and implementation in this thesis focused on fault-tolerance components, the approach in this thesis can be used in other areas where it is possible to identify independent components that need to be dynamically added. In the context of security, in [26], it is shown that a security failure is often a result of number of faults. Also, in [27], it is shown that the theory of detectors and correctors can be applied for adding fault-tolerance to Byzantine faults and these faults are often important in the context of modeling security threats. We are currently exploring techniques similar to those used in separating functionality and fault-tolerance to separate functionality and security aspects of an application. These techniques will enable us to use the approach mentioned in this paper to adaptively change these security components with changing environment conditions.

APPENDIX A

Forward Error Correction

Reliable communication protocols require that all the intended recipients of a message receive the message intact. Automatic Retransmission Request (ARQ) techniques are used in unicast protocols, but they do not scale well to multicast protocols with large groups of receivers, since segment losses tend to become uncorrelated thus greatly reducing the effectiveness of retransmissions. In such cases, Forward Error Correction (FEC) techniques can be used, consisting in the transmission of redundant packets (based on error correcting codes) to allow the receivers to recover from independent packet losses.

Forward Error Correction (FEC) is well-studied and widely used technique for reliable communication. FEC is used from application domains ranging from space communication to reliable data storage on disks. The idea of FEC is simple: given a communication channel with known error probabilities, use error correcting codes to send redundant information with a data stream, enabling the receiver to correct errors/losses, at a desired recovery rate, without contacting the sender.

Forward Error Correction is the technique by which a sender prevents loss of message by sending redundant information along with the original data, enabling the receiver to reconstruct lost data without contacting the sender. FEC techniques make extensive use of error correcting codes, which can be computationally intensive. Hence, FEC was successfully used primarily at bit-level where the unit of data involved in encode/decode operations were in the order of a few bits. Packet level

FEC was studied in [28, 29]. In [28] it is shown that packet level FEC can be implemented by transmitting M redundant packets after each set of N regular packets, so all packets can be reconstructed if at least N out of $N + M$ packets are received. Packet level forward error correction is not very difficult to implement. The network's characteristics can be used to the set the level of redundancy.

There are not many actual implementations of FEC that exist today, despite the widespread use of error correcting codes in many fields of information processing, and a general consensus on the usefulness of FEC techniques within some of the Internet protocols. The main reason being concerns related to the complexity of implementing such codes in software. In [30] the author has given a basic description of erasure codes. An implementation of a simple erasure code to be used in network protocols is described in [30]. As shown in Figure A.1, k blocks of source data are encoded at the sender to produce n blocks of encoded data, such that any k of the n encoded packets can be used to reconstruct the k source packets. The receiver can recover from up to $n - k$ losses in a group of n encoded packets.

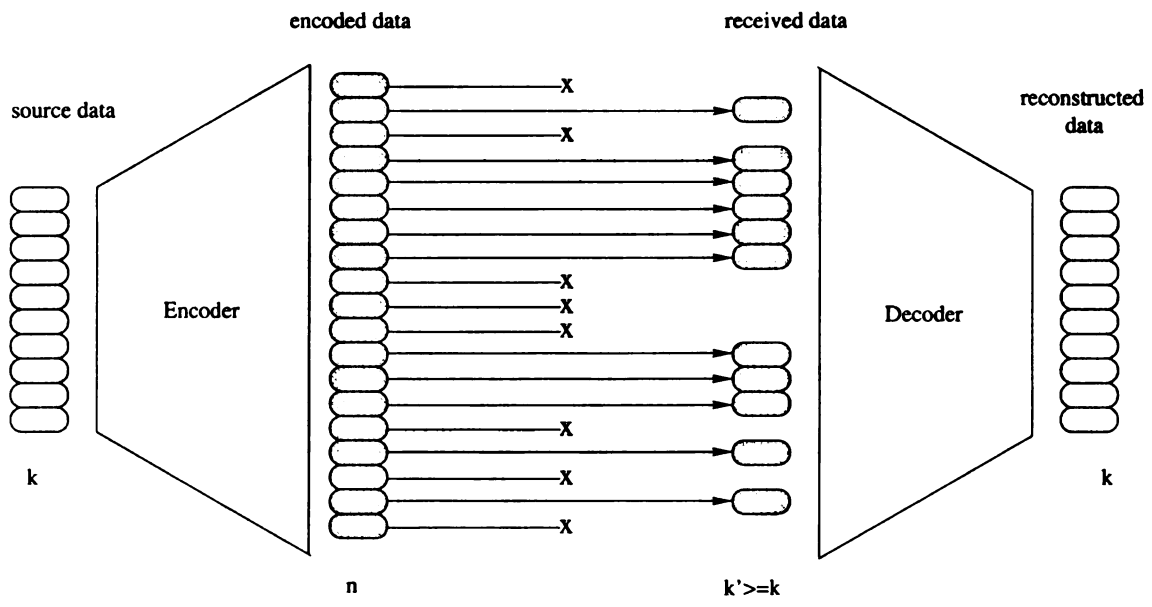

Figure A.1: Encoding/Decoding Operation of FEC. [30]

APPENDIX B

Application of the Framework for Routing

In this chapter, we discuss the application of our framework for routing in Siesta [23], a simple NEST (Network Embedded Software Technology) application simulator. Siesta is developed by the Institute of Software Integrated Systems, Vanderbilt University. It provides the flexibility of testing and experimenting the beam vibration control application that arises in a NEST scenario. It can also be used to test middleware services of other NEST systems.

B.1 Beam Vibration Control Application

The setup for the beam vibration control application is shown in Figure B.1. The application consists of 50 nodes aligned along the beam that is subjected to an outside disturbance. Each node has a sensor and an actuator. Sensors measure the point velocity of the vibration of the beam and the actuators produce point force input to counter the vibration. The objective of this application is to control the beam vibration in response to the disturbance. We consider the case where distributed control is used to control the beam vibration in Siesta. In distributed control, each node sends its sensor values to other nodes that are within a specified distance (called the *reach* of the node). Each node then calculates its actuator output based on a weighted average of its own sensor value and the sensor values it received. (The actual algorithm used to determine the actuator output is not relevant for this discussion and hence, is omitted.)

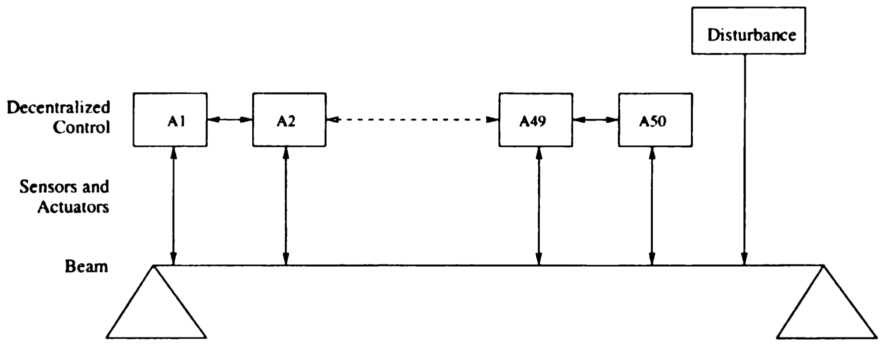

Figure B.1: Beam Vibration Control Application Setup

Siesta defines an adjacency relation among the nodes. This relation specifies the nodes that can communicate directly with each other. Communication between non-adjacent nodes is achieved using the static routing information maintained at each node. The intolerant version of the application does not deal with faults, i.e., if a node/router fails, it affects the distributed control at the nodes that are within the reach of the failed node. Hence, we focused on the design of a fault-tolerance component for routing. If a node/router fails, this fault-tolerance component routes the messages using alternate links. We describe how we specified this component and composed it with the intolerant application, next.

B.2 Fault-tolerance Component for Routing in Siesta

In Siesta, for every clock event, each node can create or execute an action. The *Router.RouteMessage* object creates an action if the router has some packets to send, by invoking the method *create*. The *create* method in the intolerant application is split into three methods: *getNextHop*, *setNextHop* and *route*. The *route* method uses *getNextHop* to determine the next hop of a message and creates an action that sends this message. Such a split makes it easier to understand the *create* method and makes it amenable for method-level synchronization.

Our framework uses the meta-application file (the contract between the framework and the intolerant application) to determine the methods that need to be trapped. This file specifies the methods that are exposed by the intolerant application to the fault-tolerance component. The meta-application file is as follows:

```
function:create
class:nest.mw.Router.RouteMessage
returntype:nest.mw.Action
arglist:nil
nexthop_get:getNextHop
nexthop_set:setNextHop
send:route
...
```

Based on the above meta-application file, our framework traps the *create* method of the *Router.RouteMessage* object. Additionally, the intolerant application also provides a method for getting the next hop of a message (*nexthop_get* field), a method for setting the next hop of a message (*nexthop_set* field) and a method for sending a message (*send* field).

We developed a fault-tolerance component, *AlternateRoute* that routes the messages in alternate links if the current next hop has failed. It requires that the intolerant application provide methods for getting the next hop of a message (*getNextHop*), for setting the next hop of a message (*setNextHop*) and for sending a message (*route*). The method information can be determined from the fields *nexthop_get*, *nexthop_set* and *send* in the meta-component file (the contract between the framework and the fault-tolerance component). Since Siesta is a simulation environment, the information about the nodes/routers that have failed is statically available to the component. However, it can be modified so that the component determines the failure of nodes/routers dynamically. If the next hop has failed, the component finds a non-failed node that is nearest to the destination. The component uses the method provided in the meta-application file of the intolerant application to change the next hop of the message. The component then calls the *route* method provided in the meta-application file of the intolerant application. The meta-component file for this component is as follows:

```
intol_function:
intol_class:
nexthop_get:
nexthop_set:
send:
ftcomponent_method:ftRoute
...
```

When this component is instantiated, our framework matches the meta-application file with the meta-component file to determine how the component and the intolerant application are composed. During dynamic addition/removal of *AlternateRoute* component, we find that the component fractions of *AlternateRoute* do not exhibit mutual dependency. Therefore, the removal of this component is simple; in the reset-transition wave, each node can remove its component fractions independently.

If a new routing component is developed for Siesta, it can also be added dynamically. Towards this end, the new component will need the corresponding meta-component file. The framework will use the meta-application file of the intolerant application and the meta-component file of the new component during instantiation. As mentioned above, the removal of *AlternateRoute* component can be achieved easily in the reset-transition wave.

BIBLIOGRAPHY

[1] A. Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems*, pages 436–443, 1998.

[2] Anish Arora and Sandeep S. Kulkarni. Designing masking fault tolerance via nonmasking fault tolerance. In *Symposium on Reliable Distributed Systems*, pages 174–185, 1995.

[3] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[4] Sandeep S. Kulkarni and Anish Arora. Multitolerance in distributed reset. *Chicago Journal of Theoretical Computer Science*, 1998.

[5] L. Lamport and L. Lynch. Distributed computing: models and methods. *Handbook of Theoretical Computer Science*, 2:1158–1199, 1990. Elsevier Science Publishers.

[6] R. Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. *Ninth ACM Data Communications Symposium*, 20(7):44–52, 1985.

[7] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computation. *Information Processing Letters*, 11(1):1–4, August 1980.

[8] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[9] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17:911–921, 1991.

[10] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7:61–77, 1989.

[11] George Varghese, Anish Arora, and Mohamed Gouda. Self stabilization by tree correction. *Chicago Journal of Theoretical Computer Science*, 1997(3), November 1997.

[12] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset. In *Workshop on Distributed Algorithms*, pages 326–339, 1994.

[13] Reinhard Wilhelm. A modified tree-to-tree correction problem. *Information Processing Letters*, 12(3):127–132, June 1981.

[14] Jean-Charles Fabre and Tanguy Perennou. FRIENDS: A flexible architecture for implementing fault tolerant and secure distributed applications. In *European Dependable Computing Conference*, pages 3–20, 1996.

[15] G. Agha, S. Frolund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Proceedings of DCCA-3*, pages 197–207, 1993.

[16] W. K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *21st International Conference on Distributed Computing Systems*, pages 635–643, April 2001.

[17] J. J. Kulzer. Systems reliability: A case study of number 4 ESS. *System Security and Reliability, Infotech State of the Art Report*, pages 186–188, 1977.

[18] Ralph Keller and Urs Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445, 1998.

[19] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA*, pages 147–155, Orlando, USA, 1987.

[20] Glen McCluskey. *Using Java reflection.* Available at: `http://developer.java.sun.com/developer/ technicalArticles/ALT/Reflection`, January 1998.

[21] Bruce Eckel. *Thinking in Java.* Prentice Hall PTR, second edition, June 2000.

[22] Ian Welch and Robert Stroud. Kava - using bytecode rewriting to add behavioral reflection in Java. In *Proceedings of USENIX Conference on Object Oriented Technology*, 2001.

[23] Akos Ledeczi, Miklos Maroti, and Istvan Bartok. *SIESTA - Simple NEST Application Simulator (Siesta v0.1 r10.31.1).* Institute for Software Integrated Systems, Vanderbilt University, Available At: `http://www.isis.vanderbilt.edu/projects/nest/downloads.asp`, October 2001.

[24] Gautam Chakrabarti and Sandeep S. Kulkarni. A modified approach to dynamic source routing in mobile ad-hoc networks. *AD-HOC Networks and Wireless Fields Institute (Toronto)*, September 2002.

[25] Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic program transformation with JOIE. In *USENIX Annual Technical Symposium*, pages 167–178, 1998.

[26] Catherine Meadows. Applying the dependability paradigm to computer security. In *Proceedings of the 1995 New Security Paradigms Workshop.* pub-IEEE, 1996.

[27] S. S. Kulkarni. *Component-based design of fault-tolerance.* PhD thesis, Ohio State University, 1999.

[28] Christian Huitema. The case for packet level FEC. In *Protocols for High-Speed Networks*, pages 109–120, 1996.

[29] P. K. McKinley, C. Tang, and A. Mani. A study of adaptive forward error correction for wireless collaborative computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):936–947, September 2002.

[30] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, April 1997.