A MODEL-BASED APPROACH FOR SELF-HEALING AND SELF-CONFIGURATION IN COMPONENT-BASED SOFTWARE SYSTEMS

by

Emad Yousif Albassam
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

________________________________	Dr. Hassan Gomaa, Dissertation Director
________________________________	Dr. Daniel A. Menascé, Committee Member
________________________________	Dr. Hakan Aydin, Committee Member
________________________________	Dr. Brian L. Mark, Committee Member
________________________________	Dr. Stephen Nash, Senior Associate Dean
________________________________	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date:___________________________	Spring Semester 2017 George Mason University Fairfax, VA

A Model-Based Approach for Self-Healing and Self-Configuration in Component-Based Software Systems

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

by

Emad Yousif Albassam
Master of Science
George Mason University, 2012
Bachelor of Science
King Abdulaziz University, 2007

Director: Hassan Gomaa, Professor
Department of Computer Science

Spring Semester 2017
George Mason University
Fairfax, VA

DEDICATION

To Allah for giving me the strength to carry this work
To my wife, daughter, and parents for their unwavering support and encouragement

ACKNOWLEDGEMENTS

First and foremost, I am deeply grateful to my dissertation director Dr. Hassan Gomaa for his guidance, support, and encouragement throughout the dissertation process. I am also grateful to Dr. Daniel. A. Menascé for his invaluable advice and help throughout the course of this research. I would also like to thank Dr. Hakan Aydin and Dr. Brian L. Mark for serving in my committee and for their guidance. I would also like to thank my research colleague Jason Porter for his frequent help and support during this research.

I would like to express my warmest and deepest appreciation to my wife Halah Alkadi for her unconditional love, support, and patience during this process. She managed to keep me motivated even though she is pursuing her own doctorate. I am also deeply thankful to my daughter Jouri who managed to keep me joyful when I needed it the most. I’m also deeply grateful to my parents, Yousif and Wafa, who without them, I would have never accomplished my dreams.

Finally, I am thankful to King Abdulaziz University and my country for their scholarship to pursue this degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A MODEL-BASED APPROACH FOR SELF-HEALING AND SELF-CONFIGURATION IN COMPONENT-BASED SOFTWARE SYSTEMS

Emad Yousif Albassam, Ph.D.

George Mason University, 2017

Dissertation Director: Dr. Hassan Gomaa

Component-based software architectures (CBSAs) are a well-known approach for building increasingly complex software systems from components that are intended to be distributed and autonomic. However, CBSAs often run in environments that are evolving and subject to failures. As a result, it is highly desirable to design CBSAs with self-configuration and self-healing capabilities so that they can dynamically adapt and recover in response to changing environments and failures, where the goal is to minimize manual intervention involved in managing and evolving these architectures. However, the systematic integration of the self-healing and self-configuration properties remains a challenge. Furthermore, although there exist a large body of literature in the areas of self-healing and self-configuration, most of them use a centralized approach. The main challenge with decentralized approaches is carrying out dynamic adaptation and recovery using partial knowledge of the system.

This dissertation describes the design of Recovery and Adaptation Connectors. A Recovery and Adaptation Connector (RAC) extends communication connectors in CBSAs so that in addition to managing communications between application components, a RAC also manages adaptation and recovery concerns of these components. Each RAC encapsulates an Adaptation and Recovery State Machine that defines the behavior of the RAC during (1) normal execution when there are no adaptation or failures, (2) recovery so that the RAC ensures that any transactions that were interrupted due to a run-time failure are recovered and restarted at the recovered component, and (3) dynamic adaptation so that a component is only adapted after it has completed all transactions that it is currently engaged in and has become quiescent.

In addition, this dissertation describes the design of a decentralized framework called DARE for providing CBSAs with both self-healing and self-configuration properties. DARE integrates architecture discovery mechanisms with recovery and adaptation connectors, and its design is based on a decentralized MAPE-K loop model in which DARE carries out recovery and dynamic adaptation when only partial knowledge of the software system is known to each node.

1 INTRODUCTION

This dissertation investigates a reuse, model-based approach for self-configuration and self-healing in component-based software architectures (CBSAs) that enables software systems to dynamically adapt and recover in response to changing environments and failures. The goal is to minimize manual intervention involved in managing and evolving these architectures. A CBSA with the self-configuration property has the capability of automatically adding, removing, and replacing components seamlessly at run-time. On the other hand, a CBSA with the self-healing property is capable of detecting and recovering from failures by dynamically relocating failed components to different nodes and establishing a consistent state in order to resume normal execution.

This chapter is organized as follows. Section 1.1 discusses the motivation of this dissertation. Section 1.2 provides a glossary of terms that are used throughout this dissertation. Sections 1.3 and 1.4 contain the problem statement and research hypothesis, respectively. Section 1.5 enumerates the main objectives of this dissertation. Section 1.6 lists the assumptions of this dissertation. Section 1.7 contains the organization of the remainder of this dissertation.

1.1 **Motivation**

Manual management of large and highly dynamic CBSAs is becoming increasingly difficult as the size and complexity of these systems increase. For instance, studies showed that the mean time to failure in high-performance computing systems that consist of thousands of computational nodes is a few hours (Cappello et al., 2014; Schroeder and Gibson, 2007). As a result, approaches based on autonomic computing have gained attention for developing systems that are capable of self-configuration, self-healing, self-optimization, and self-protection, i.e., exhibit self-* properties. However, the systematic integration of these self-* properties is one of the main challenges of autonomic computing (Kephart and Chess, 2003).

In addition, although software adaptation and recovery techniques are widely used to evolve CBSAs at run-time and to cope with failures in distributed software systems, it would be beneficial to apply *reuse* concepts to these techniques since reuse is a desirable feature in software development. Several reuse concepts and approaches have been successfully applied, such as reusing existing components or entire software architectures, to speed up development time and increase dependability in software systems (Sommerville, 2010).

Two means that are widely used to achieve software reuse are *software patterns* and *software product line* (SPL) technology. Software patterns define solutions to recurring problems in software design. Several kinds of patterns addressing different kinds of problems were developed over the last decades such as *design patterns* (Gamma et al., 1994), *architectural patterns* (Taylor et al., 2009), and *adaptation patterns* (Gomaa and Hussein, 2004). On the other hand, the goal of SPL technology is to design a family of software systems that share some commonality by designing a reusable software architecture that can be tailored to derive each member of the SPL family (Clements and Northrop, 2001).

This dissertation investigates the design of a decentralized framework that applies reuse concepts, including software patterns and SPL technology, to software adaptation and recovery and integrates both self-healing and self-configuration in order to minimize manual intervention involved in managing CBSAs.

1.2 Glossary of Relevant Terms

This section provides a glossary of recurring terms that are used throughout this dissertation:

- **Adaptation Pattern.** A software adaptation pattern defines how a set of components that make up an architectural pattern dynamically cooperate to change the software configuration to a new configuration (Gomaa and Hussein, 2004).
- **Architectural Pattern**. A recurring software architecture that can be used in a variety of software applications (Gomaa, 2011).
- **Autonomic Controller.** A control component used to automate management of distributed software systems by providing the following autonomic properties: self-healing, self-configuration, self-optimization, and self-protection (Kephart and Chess, 2003).
- **Configuration Map**. A software artifact that describes deployment of the software systems in terms of components, nodes, and mapping between components to nodes (Taylor et al., 2009).

- **MAPE-K Loop Model.** A widely used model to implement autonomic controllers that consists of four activities (monitoring, analysis, planning, and execution) that operate on a knowledge-base of the system (Kephart and Chess, 2003).
- **Message-Based Transactions.** A transaction in CBSAs is defined as an information exchange between multiple components through messages (Kramer and Magee, 1990) while a transaction in transactional processing systems is defined as an atomic unit of work (Bernstein and Newcomer, 2009). This dissertation combines these two definitions as: a transaction is an information exchange between two or more components through messages such that either all messages in a transaction are eventually exchanged or none of them are.
- **Recovery and Adaptation Connector (RAC).** RACs extend connectors in CBSAs with recovery and adaptation capabilities to assist in self-healing and self-configuration.
- **Recovery Pattern.** A recovery pattern defines how components in an architectural pattern can be dynamically relocated and recovered to a consistent state after a component has failed.
- **Self-Configuration**. The ability of the software system to automatically adapt its architecture by adding, removing, or replacing components seamlessly at run-time in response to changes in operational environment or user requirements (Kephart and Chess, 2003).
- **Self-Healing**. The ability of the software system to automatically detect failures and then recover to a consistent state so that it can resume normal execution (Kephart and Chess, 2003).

- **Software Architecture**. A software artifact that describes the overall structure of the software system in terms of components and their interconnections using connectors (Taylor et al., 2009).
- **Software Product Line**. A family of software systems whose members share some commonality but also have variable functionality (Clements and Northrop, 2001).

1.3 Problem Statement

There are no existing approaches for self-configuration and self-healing for handling recovery and dynamic adaptation of component-based software architectures that take into consideration the architectural structure patterns used in a system and the architectural communication patterns between the system's components.

1.4 Research Hypothesis

It is possible to design a decentralized approach that integrates both self-configuration and self-healing in component-based software systems such that reusable adaptation and recovery patterns can be used to dynamically adapt the software architecture as well as determine the precise recovery actions to restore the system back to a consistent state after a run-time failure so that it can resume normal execution.

1.5 Research Objectives

The main objectives of this dissertation are to describe the design of:

1. *Recovery and adaptation patterns*. This dissertation investigates the design of adaptation and recovery patterns that define how components in an architectural pattern can be dynamically adapted or recovered to a consistent state after a run-time failure.

2. *Recovery and adaptation connectors*. This dissertation shows how connectors in component-based software architectures can be extended with adaptation and recovery capabilities to assist in self-healing and self-configuration, where the goal is to separate adaptation and recovery concerns from the business logic carried out by application components.
3. *A decentralized, self-healing and self-configuration framework*. This research describes the design of an architecture-based, decentralized framework for self-healing and self-configuration that is responsible for carrying out dynamic adaptation and recovery of components after a run-time failure.
4. *A reusable recovery and adaptation connector.* This dissertation shows how variability in architectural patterns can be managed by designing a reusable recovery and adaptation connector using software product line technology.

1.6 **Assumptions**

This dissertation makes the following assumptions:

- Only one node fails at a time.
- Failures are not caused by malicious attacks or buffer overflows and follow a fail-stop model.
- Failure does not occur during recovery or adaptation.
- Assistant Recovery and Adaptation Connectors do not fail.
- Message delivery uses a reliable network transport protocol.

1.7 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 discusses the related works to this dissertation. Chapter 3 describes recovery and adaptation patterns in service-oriented architectures (SOAs) and the design of the recovery and adaptation connector that can be used to separate recovery and adaptation concerns from service and coordination concerns. Chapter 4 discuss recovery and adaptation patterns for various asynchronous patterns and the corresponding design of the recovery and adaptation connector for these patterns. Chapters 5 describes the DARE framework, which is an architecture-based, decentralized framework for providing both self-healing and self-configuration properties to large and highly dynamic CBSAs. Chapter 6 describes the design of an assistant recovery and adaptation connector for handling recovery and adaptation concerns of clients and producers. Chapter 7 describes the approach of recovering the recovery and adaptation connector after a run-time failure. Chapter 8 describes how a reusable recovery and adaptation connector can be designed using Software Product Line technology. Chapter 9 defines formal properties of the approach. Chapter 10 describes the experimental validation and results. Chapter 11 concludes this dissertation and discusses the future work.

2 RELATED WORK

This chapter discusses previous research efforts that relate to this dissertation. Section 2.1 provides a description of autonomic software systems and the self-* properties. Section 2.2 discusses related surveys on autonomic software systems and describes how some of the open challenges in this area are tackled by this dissertation. Section 2.3 describes related dynamic software adaptation techniques and self-configuration frameworks. Section 2.4 provides an overview of software recovery and fault-tolerance techniques, and describes related self-healing frameworks. Section 2.5 describes related works in the area of software product lines.

2.1 Autonomic Software Systems

Autonomic software systems are software systems that exhibit the following properties (Kephart and Chess, 2003):

- *Self-configuration:* the ability for the software system to automatically change its configuration based on high-level policies.
- *Self-healing*: the ability for the software system to automatically detect failures, diagnose the cause of the failure, and install the necessary repairs to recover from the failure.

- *Self-optimization*: the ability for the software system to automatically select the optimal operational parameters to improve the quality of its services based on the current context.
- *Self- protection:* the ability for the software system to automatically react to malicious attacks by executing defensive and prevention actions.

In autonomic software systems, an autonomic manager is configured with one or more high-level objectives and is attached to one or more managed elements (e.g. a hardware or software component) to provide them with autonomic behavior. An autonomic manager executes a MAPE-K control loop that consists of the following activities (1) *Monitoring* the managed elements and collecting various relevant information (e.g. response time or CPU utilization), (2) *Analyzing* the collected information and asserting that no high-level objectives are violated, (3) *Planning* for behavioral or structural changes if one or more high-level objectives are violated, (4) *Executing* the required changes in order to restore the system back to a state that satisfies all high-level objectives. Additionally, autonomic managers are embedded with the necessary *Knowledge* about the managed elements required by the MAPE activities.

2.2 **Autonomic Software Systems Challenges and Surveys**

Although there exists a large body of literature in the area of autonomic and self-adaptive systems, the second road map of self-adaptive systems stated that the focus of the majority of the works in this area is on centralized approaches (Lemos et al., 2013). The main challenge with decentralized approaches is carrying out dynamic adaptation and recovery using partial knowledge of the software system (Krupitzer et al., 2015).

This dissertation focuses on a decentralized approach for self-healing and self-configuration by assuming that none of the nodes has the complete knowledge of the software system.

Weyns et al. stated that there is a need to further study possible decentralization patterns for the MAPE-K control loop (Weyns et al., 2013). With that respect, this dissertation investigates the design of a self-healing and self-configuration framework that is based on a decentralized version of the MAPE-K loop model and shows how coordination can be achieved between the various managers that realize the decentralized MAPE-K loop model.

Schneider et al. (Schneider et al., 2015) concluded in their survey on self-healing frameworks that the systematic integration of the self-* properties is one of the main challenges in this area. This dissertation investigates this problem by describing the design of a decentralized framework that provides large and highly dynamic CBSAs with both self-healing and self-configuration properties.

Psaier and Dustdar surveyed self-healing approaches and showed that these approaches are dependent on the application domain (Psaier and Dustdar, 2010). This dissertation considers how a self-healing and self-configuration approach can be designed independently of the application domain by considering the architectural patterns involved in the application. Similarly, Salehie and Tahvildari (Salehie and Tahvildari, 2009) surveyed approaches to self-adaptive systems and proposed a taxonomy for the major design considerations that are often associated with these systems. They stated that isolating problematic components and recovering components correctly after a failure is a challenge. This dissertation tackles these issues by considering extending connectors in CBSAs with adaptation and recovery capabilities so that adaptation and recovery concerns of a component are localized to the connector of that component. Neti and Muller (Neti and Muller, 2007) identified the main challenge in self-healing systems as the ability for a system to determine the cause of failure and then recover correctly. This dissertation investigates the design of recovery patterns that enable CBSAs to determine failed transactions and the precise actions to recover the system to a consistent state after a run-time failure.

Kramer and Magee (Kramer and Magee, 2007) identified several challenges involved in dynamic software adaptation including (1) preserving the consistency of a software system and (2) ensuring that no state information is lost during reconfiguration. Similarly, a survey by Huebscher and Mccann (Huebscher and McCann, 2008) stated that adapting software systems correctly at run-time without causing undesirable behavior remains a challenge in this area. This dissertation tackles these problem by considering how adaptation patterns can be used to adapt CBSAs seamlessly at run-time without losing any state information.

2.3 Self-Configuration

In the area of self-configuration, Garlan et al. described Rainbow, a reusable, architectural-based framework for self-adaptive software systems that is based on the MAPE-K control loop (Garlan et al., 2004). Rainbow maintains runtime information about a system’s structure and attaches various attributes to components and connectors that are used at runtime as constraints for triggering adaptation. Whenever a constraint is violated, Rainbow executes an adaptation strategy (Cheng and Garlan, 2007) to restore the application back to a state that satisfies all constraints.

Menasce, Gomaa, Malek, and Sousa described SASSY, an architectural-based adaptation framework for service-oriented architectures (SOAs) that realizes the MAPE-K control loop, and showed how it can re-architect SOAs at runtime by finding a near-optimal software configuration using quality of service (QoS) architectural patterns whenever the utility of the system falls below a threshold (Menasce et al., 2011, 2010). MOSES (Cardellini et al., 2012) is a framework aimed to improve the QoS attributes of SOAs in which an optimization engine is used to compute policies for manipulating response time, availability, and cost attributes of SOAs at run-time.

Kramer and Magee (Kramer and Magee, 2007) investigated a decentralized change manager that maintains a complete view of the software system by relying on reliable broadcasting and totally ordered message delivery. MUSIC (Hallsteinsen et al., 2012) is a framework for developing context-aware, self-adaptive software systems. However, recovery of the application state after a failure is assumed to be done at the application level. Bisadi and Sharifi (Bisadi and Sharifi, 2009) discussed an architecture that is inspired by cellular adaptation in which connectors forward incoming requests to a healer component for further analysis. The healer component then uses application-specific policies to determine the need for adaptation. However, their approach considers adaptation caused by (1) increased number of messages (i.e. load) at connectors which requires increasing number of components to handle these messages and (2) incompatible message types which require searching for and installing compatible components.

Kramer and Magee investigated how components must transition to a quiescent state to safely reconfigure a software system while it is operational (Kramer and Magee, 1990). Vandewoude et al. investigated relaxing the quiescence requirement (Vandewoude et al., 2007). Ramirez and Cheng describe patterns for self-adaptive systems including patterns for inserting and removing components, reconfiguring service components, and reconfiguring decentralized architectures (Ramirez and Cheng, 2010). Li et al. (Li et al., 2006) showed how connectors can be used to dynamically compose services in service-oriented architectures (SOAs).

Gomaa et al. investigated dynamic software adaptation patterns that define how components in an architectural pattern can dynamically collaborate at run time to change the current configuration of the system to a new configuration (Gomaa, Hashimoto, Kim, Malek, and Menasce, 2010; Gomaa and Hashimoto, 2012; Gomaa and Hussein, 2004). In their approach, adaption state machines are embedded inside adaptation connectors (rather than application components) in order to increase the reusability of adaptation and to allow connectors to queue incoming messages while components are being adapted so that normal operation can be resumed after dynamic adaptation has completed.

Although several reference architectures and reusable frameworks have been proposed to achieve self-configuration (Dashofy et al., 2002; Garlan and Schmerl, 2002; Oreizy et al., 1999), these approaches do not consider the self-healing property.

2.4 Self-Healing and Software Recovery Techniques

2.4.1 Frameworks

Prior works on self-healing approaches vary based on the kind of problems assumed to occur in the software system. For instance, several approaches have been prescribed for handling software aging and transient faults (Silva et al., 2009), performance degradation (Magalhães and Silva, 2015), and software faults (Bruning et al., 2007). We focus here on related works that are capable of handling failures under the fail-stop model in which components do not behave erroneously but simply cease functioning when they fail (Avizienis et al., 2004).

Stojnic and Schuldt described OSIRIS-SR, a decentralized, scalable safety ring to achieve self-healing data management in service-oriented architectures (Stojnic and Schuldt, 2012). However, their approach does not handle dynamic deployment of components. Danilecki et al. (Danilecki et al., 2011) described the use of ReServE to recover services in SOAs to a consistent state after a run-time failure. However, their approach does not consider dynamic adaptation of the software architecture. Prior works showed how BPEL can be extended with self-healing capabilities (Modafferi and Conforti, 2006; Subramanian et al., 2008) and considered multiple recovery strategies for recovering web services (Angarita et al., 2016). Salatge et al. suggested the use of fault-tolerance connectors to increase service dependability in SOAs (Salatge and Fabre, 2007). However, none of these works considered integrating self-configuration, by driving the state of components to a quiescent state, with self-healing capabilities. Although it is possible to design platform-dependent self-healing approaches (e.g. Candea et al. showed how JBoss can be extended with self-healing capabilities (Candea et al., 2003)), this dissertation considers a platform-independent self-healing approach to increase reuse.

2.4.2 Software Recovery

Software recovery is concerned with techniques that enable the restoration of software state to a consistent state after an error has occurred. This section discusses software recovery techniques that relate to this dissertation including recovery in transactional processing systems, transactional queues, and roll-back recovery techniques.

Recovery in transactional processing systems depends on the notion of transactions. A transaction (Bernstein and Newcomer, 2009) is defined as a logical unit of work that consists of two or more operations such that either all operations execute to their entirety (in which case the transaction is committed) or none of them do (in which case the transaction is aborted). To facilitate recovery, transactional processing systems maintain log files to keep track of transaction statuses and the changes made by each transaction to the system state. Recovery in these systems involves reconstructing a consistent state (Bernstein and Newcomer, 2009; Lomet and Tuttle, 2003; Mohan et al., 1992) such that this state includes only the effect of transactions that committed before the failure. That is, the system recovers by aborting all active transactions that did not complete before the failure and storing the results of all transactions that have been logged as committed before failure.

Transactional queues (Bernstein et al., 1990) are a widely used recovery approach in distributed environments where the goal is to ensure that every request from a client to a service is processed by the service exactly once. In this approach, a client queues a request into a server’s transactional queue using a transaction. The server then dequeues the request from its transactional queue, processes the request, and queues the response into the client’s transactional queue using a second transaction. Finally, the client dequeues the response from its transactional queue and processes the response using a third transaction. Thus, transactional queues ensure that the state of the queues can be restored in case of failure.

In roll-back recovery (Elnozahy et al., 2002), distributed software processes write messages they receive or send in log files, so that these messages can be replayed during recovery after a process has failed in order to restore that process to the closest state to the failure point. Three types of message logging protocols exist in the literature (Alvisi and Marzullo, 1998; Elnozahy et al., 2002). In the *pessimistic message logging protocols*, messages are synchronously logged before they are processed, imposing some overhead during normal execution due to logging but in favor of avoiding orphan processes (i.e. processes whose state depends on a message that has been lost due to failure). In *optimistic message logging protocols* (Strom and Yemini, 1985), messages are asynchronously logged while they are being processed, which minimizes logging overhead during normal execution but may require rolling back the state of multiple processes, which complicates recovery. Finally, in *causal message logging protocols* (Lee et al., 1998), each process piggybacks the messages that affected its state when sending messages to other processes, so that these messages are logged by the recipient processes. A combination of these protocols is also possible. For instance, Wang et al.

(Wang et al., 2007) discussed combining the pessimistic and optimistic message logging in service components. In their approach, logging of messages between services from the same service provider is optimistic while logging of messages between services from different service providers is pessimistic.

This dissertation considers how messages exchanged between components in an architectural pattern can be treated as atomic transactions such that these transactions can be recovered by recovery and adaptation connectors after a run-time failure. Furthermore, this dissertation considers how these connectors can be recovered after a run-time failure use message logging and replaying.

2.4.3 Fault Tolerance

To gain better understanding in fault tolerance concepts and techniques, several surveys and references are used as follows. Work by Avizienis et al. (Avizienis et al., 2004) discussed important concepts related to dependable systems including faults, errors, and failures and their classifications as well as fault tolerance techniques. Guerraoui and Schiper (Guerraoui and Schiper, 1997) described techniques related to replication of software components. Freiling et al. (Freiling et al., 2011) surveyed failure detection techniques. Raynal (Raynal, 1992) surveyed techniques for logical clock synchronization. Holman and Lee (David M Holman, 2008) discussed algorithms to achieve network fault tolerance including the store-and-forward technique. Fault Tree Analysis (FTA) is a top-down approach introduced by Bell Laboratories in 1962 to evaluate safety and reliability properties in computer systems (Ericson, 1999). In FTA, a fault tree model is used to depict the relationship between a high-level, undesirable system event (located at the top of the tree) and the basic cause events (located at the leaves of the tree) that could trigger the undesirable event (Leveson and Harvey, 1983). The relationship between cause events in fault trees is defined using logical gates. Furthermore, an event at one level in a fault tree is resolved by identifying its immediate cause events in the next level until the basic cause events are identified or no further resolution is necessary. Once a fault tree is defined, a qualitative evaluation is possible through Boolean algebra to find the combinations of minimum basic cause events (called minimum cut sets) that can cause the top event (Rauzy, 1993).

In order to support failure recovery and dynamic adaptation, this dissertation considers how a recovery and adaptation connector can be designed so that its stores the messages it receives in queues before forwarding them to their destination so that these messages can be recovered in case of failure.

2.5 Software Product Lines and Dynamic Software Product Lines

A software product line (SPL) is a family of software systems that share some commonalities and have some differences (Clements and Northrop, 2001). SPL design methods and techniques deal with developing reusable software assets, including a feature model and a software architecture, that can be tailored at deployment time to generate a particular member of the SPL. A common approach in these methods is to design a feature model that captures the commonality and variability, in terms of functional and non-functional requirements, among the SPL members as well as any constraints. An example of a SPL design method is the Product Line UML-based Software Engineering (PLUS) (Gomaa, 2004). In PLUS, variability among members of a

SPL is captured by a feature model. During deployment, a member of a SPL is derived from a reusable software architecture by selecting the required features from the feature model and the corresponding components that implement those features.

Dynamic software product lines (DSPL) are SPLs that are capable of changing one member of the SPL to another while the system is operational (Hinchey et al., 2012). The feature model itself may evolve at runtime due to unanticipated variability occurring at run-time (Bencomo et al., 2012). Work by Bosch and Capilla (Bosch and Capilla, 2012) discusses the use of types and supertypes in feature models to handle dynamic evolution of these models as well as possible rebinding mechanisms in DSPL. Baresi, Guinea, and Pasquale (Baresi et al., 2012) investigated applying DSPL to service-oriented systems. In their approach, a CVL (Common Variability Language) library is used to define core and additional process elements. During execution, services are intercepted using aspect-oriented programming and required changes are applied based on dynamic feature selection and the placement or replacement of process elements. Sawyer, Mazo, Diaz, Salinesi, and Hughes (Sawyer et al., 2012) investigated generating a SPL variant based on the current operational context by combining goal-modeling and constraint programming. Gomaa and Hashimoto (Gomaa and Hashimoto, 2011) discussed extending the PLUS method to handle dynamic adaptation for service-oriented product lines. In their approach, a member of a service-oriented product line can be adapted at run-time by 1) dynamically activating/deactivating features using a run-time feature model 2) determining how the target architecture needs to be adapted as a result of dynamic feature selection by using a run-time feature/component dependency table and 3) dynamically adapting the architecture using service-oriented adaptation patterns.

3 DESIGN OF RECOVERY AND ADAPTATION CONNECTORS FOR SERVICE-ORIENTED ARCHITECTURES

This chapter describes the design of the basic structure of a recovery and adaptation connector (RAC) for service-oriented architectures (SOAs). We assume that there are multiple clients and a single service that processes multiple client requests concurrently. The service responds to each request from the client. The RAC manages transactions between a client and a service that comprise either single request/response messages or a dialog. This chapter also shows how the same RAC design can handle adaptation and recovery in other, more complex architectural patterns.

This section is organized as follows. Section 3.1 describes the design of the RAC for handling adaptation and recovery of stateless services. Section 3.2 shows how this design of the RAC can be extended to handle stateful services. Section 3.3 shows how the RAC design can be used to handle adaptation and recovery in different SOA patterns. This chapter also discusses how the RAC can use the Two-Phase Commit protocol to handle services with non-idempotent operations (section 3.2), SOA (section 3.3.4), and distributed transactions (3.3.5).

3.1 Design of the Service Recovery Connector

The RAC (Figure 3.1) behaves as a proxy for the service by receiving requests from clients and then forwarding these requests to the service. The RAC also receives responses from the service, which are then forwarded to requesting clients.

To ensure safe adaptation at run-time and recoverability of service failures, the RAC must keep track of the transactions that the service is currently engaged in and must maintain messages (i.e., requests and responses) that pass through it so that these messages can be held during adaptation and can be recovered when the service fails.

The RAC has a control object (Connector Control in Figure 3.1) that handles sending messages to and receiving responses from application components, and also handles adaptation and recovery concerns of the service. To facilitate maintenance of application messages, requests and responses are stored by the RAC in queues located at the Service Request Manager and the Service Response Manager (Figure 3.1), respectively. Each manager is provided with a coordinator component for controlling the queues it manages. The goal of these coordinators is to separate the concerns of queue management from adaptation and recovery concerns handled by Connector Control.

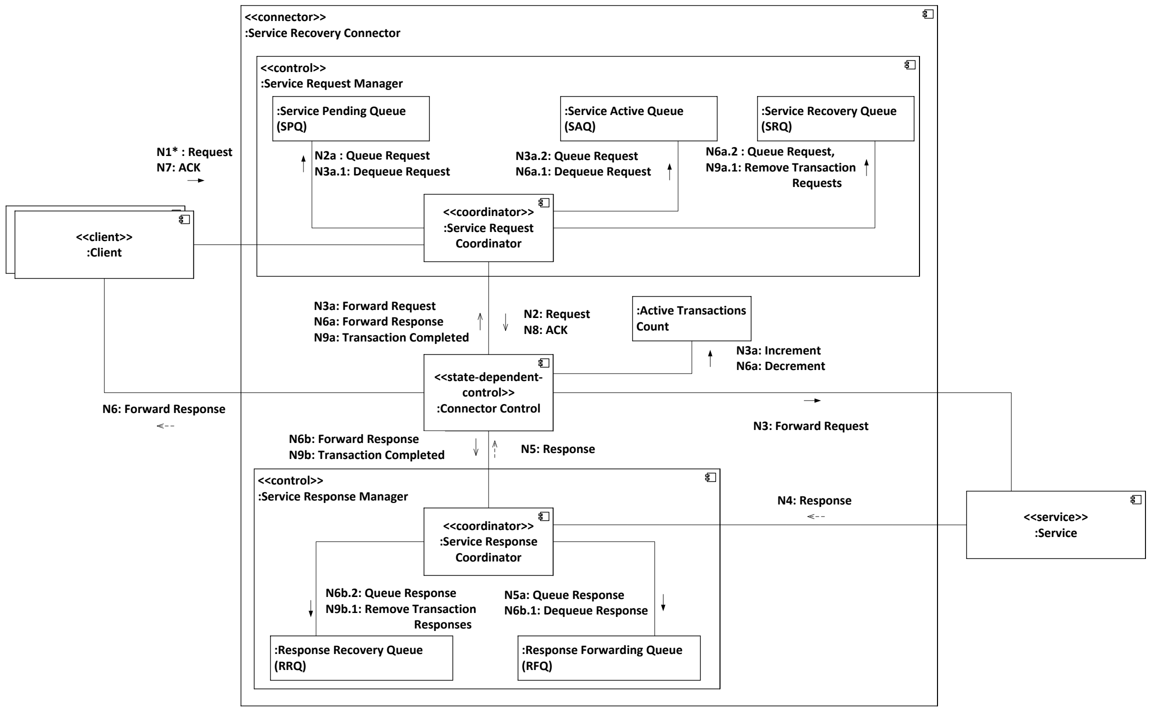


Figure 3.1 Design of service recovery connector showing messages during normal execution

3.1.1 Service Request Manager

Every request sent by a client to a service passes through the Service Request Coordinator (Figure 3.1). The Service Request Coordinator maintains the following three queues for storing client requests based on the status of these requests:

Service Pending Queue (SPQ). The SPQ stores client requests received by the RAC but that have not yet been forwarded to the service. The purpose of this queue is to buffer requests for the service so that any requests received by the RAC while the service is being dynamically adapted or is in the failed state are preserved until the service becomes active again. Thus, the SPQ ensures that no requests to the service are lost due to dynamic adaptation or recovery.

Service Active Queue (SAQ). This queue stores client requests that have been forwarded to the service but do not have corresponding service responses at the RAC, either because the service is still processing the request and has not generated the corresponding response yet or because the service response was lost due to service failure.

The RAC uses this queue to determine pending requests that must be processed by the service first before the service can be dynamically adapted. Furthermore, the RAC uses this queue to recover requests that were lost by the service (due to service failure) before the corresponding responses of these requests are received by the RAC.

Service Recovery Queue (SRQ). This queue stores client requests that have corresponding service responses at the RAC. This queue ensures that previous requests of each dialog that the service is currently engaged in are preserved so that these dialogs can also be recovered in case they were interrupted due to service failure.

3.1.2 Service Response Manager

Responses sent by the service are received by the Service Response Coordinator (Figure 3.1). The Service Response Coordinator maintains two queues for storing responses:

Response Forwarding Queue (RFQ). This queue stores service responses that have been received by the RAC but have not yet been forwarded to the requesting client.

Response Recovery Queue (RRQ). This queue stores service responses after they have been forwarded to the requesting clients. This queue ensures that a service response that has been forwarded by the RAC to the requesting client cannot be lost due to client failure. In this case, when the RAC receives a duplicate request from a recovered client, the corresponding response is obtained from the RRQ and forwarded to the recovered client, without requiring the service to process the request again.

3.1.3 Connector Control State Machine

Connector Control (Figure 3.1) is a state-dependent control component that handles recovery and adaptation of the service by tracking its current state. While the service is active, Connector Control keeps track of whether the service is currently engaged in any transactions with its clients so that it can base its adaptation and recovery decisions accordingly.

The Connector Control state machine (Figure 3.2) consists of two orthogonal state machines (STMs). Integrated Adaptation and Recovery is the orthogonal STM that handles service adaption and recovery. The Message Queue Management STM is responsible for notifying the Service Request Coordinator and the Service Response Coordinator when a client acknowledges the completion of a transaction to enable these coordinators to remove the messages of this transaction from their queues.

The orthogonal integrated adaptation and recovery state machine (Figure 3.3) consists of three composite states: (1) Active, which defines behaviour during normal service execution, (2) Adapting, which defines behaviour during dynamic service adaptation, and (3) Recovering, which defines behaviour during recovery.

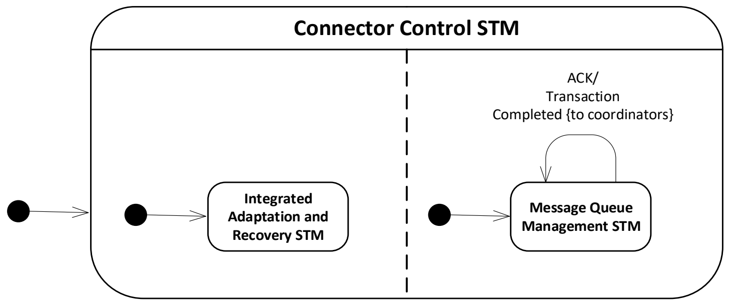


Figure 3.2 State machine executed by Connector Control

3.1.4 Normal Service Execution

Initially, Connector Control is in the Waiting for Request state (Figure 3.3) indicating that the service is currently not engaged in any transactions with its clients. When Connector Control receives a client request, it forwards the request to the service, increments the number of active transactions that the service is currently engaged in, and transitions to the Processing state. While in the Processing state, Connector Control forwards requests to the service and forwards responses to requesting clients. Connector Control remains in the Processing state as long as the service is engaged in one or more transactions. Furthermore, Connector Control increments the number of active transactions when it forwards a request that initiates a new transaction with the service and decrements this number when it receives the final response of a transaction from the service. At this time, Connector Control forwards that final response of the final transaction to the requesting client and transitions back to the Waiting for Request state.

3.1.5 Dynamic Service Adaptation

In order to safely adapt the service at run-time, the service must be in a quiescent state (Kramer and Magee, 1990) in which it is not involved in any transactions and will not receive any new transactions from its clients. At this point, the service can be removed or replaced at run-time after it has sent the final response of every transaction it is currently engaged in. In the Passivating state, Connector Control must not forward any requests that initiate new transactions with the service, so that the service can eventually transition to the quiescent state where it can be safely adapted.

If Connector Control receives the Passivate command from Change Management (Kramer and Magee, 1990) while it is in the Waiting for Request state (Figure 3.3), then the service is not engaged in any transactions with its clients. It thus transitions immediately to the Quiescent state, and notifies the Service Request Coordinator that the service is quiescent so that it holds all requests it receives from clients in the SPQ. On the other hand, if Connector Control receives the Passivate command while it is in the Processing state, then the service is engaged in one or more transactions with its clients. In this case, Connector Control transitions to the Passivating state, where the service completes existing transactions. While in the Passivating state, Connector Control forwards intermediate requests it receives to the service and forwards service responses it receives to requesting clients. Eventually, when all active transactions are completed, Connector Control notifies the Service Request Coordinator that the service is transitioning to the Quiescent State where the service can be safely adapted.

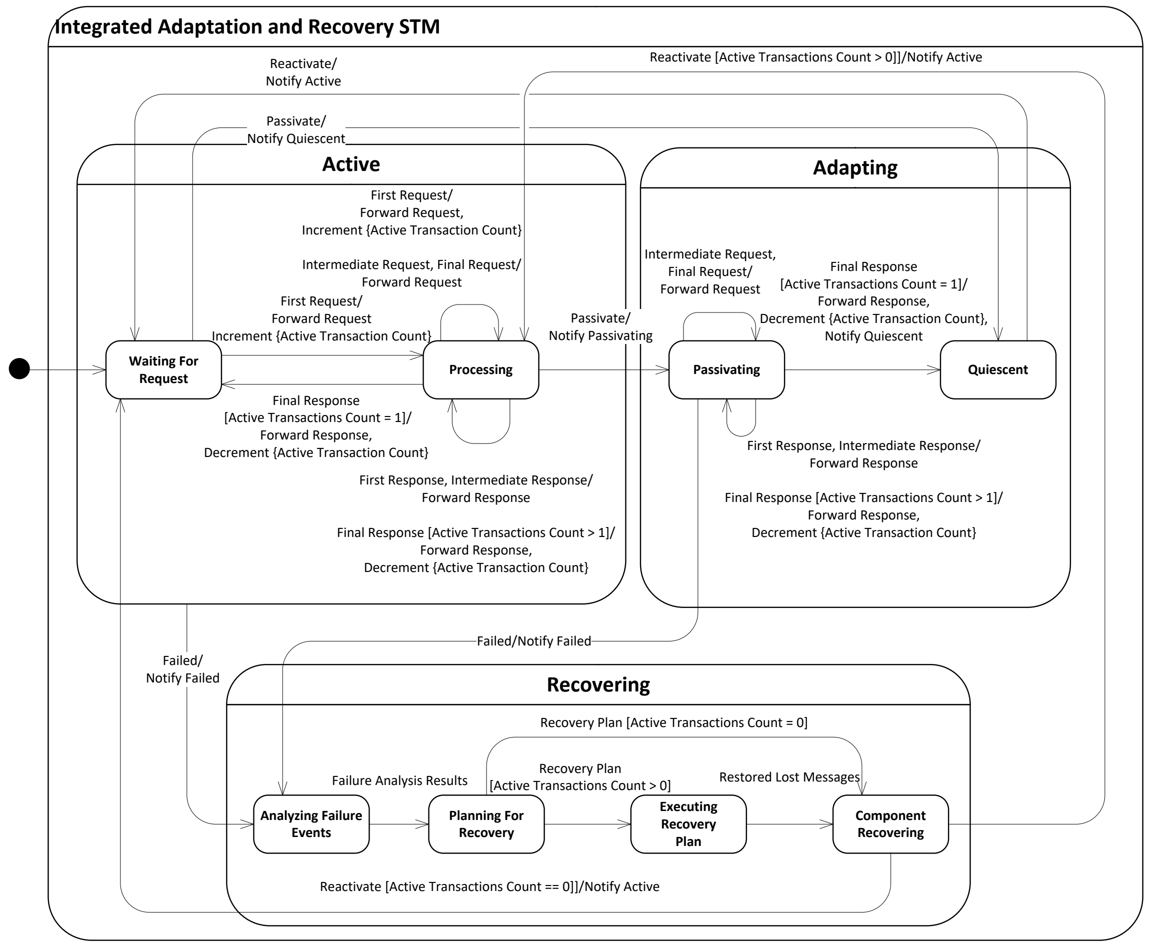

Integrated Adaptation and Recovery STM
Reactivate/
Notify Active
Reactivate [Active Transactions Count > 0]]/Notify Active
Passivate/
Notify Quiescent
Active
Adapting
First Request/
Forward Request,
Increment {Active Transaction Count}
Intermediate Request, Final Request/
Forward Request
First Request/
Forward Request
Increment {Active Transaction Count}
Waiting For Request
Processing
Final Response
[Active Transactions Count = 1]/
Forward Response,
Decrement {Active Transaction Count}
First Response, Intermediate Response/
Forward Response
Final Response [Active Transactions Count > 1]/
Forward Response,
Decrement {Active Transaction Count}
Passivate/
Notify Passivating
Intermediate Request,
Final Request/
Forward Request
Final Response
[Active Transactions Count = 1]/
Forward Response,
Decrement {Active Transaction Count},
Notify Quiescent
Passivating
Quiescent
First Response, Intermediate Response/
Forward Response
Final Response [Active Transactions Count > 1]/
Forward Response,
Decrement {Active Transaction Count}
Failed/Notify Failed
Failed/
Notify Failed
Recovering
Recovery Plan [Active Transactions Count = 0]
Recovery Plan
[Active Transactions Count > 0]
Failure Analysis Results
Restored Lost Messages
Analyzing Failure Events
Planning For Recovery
Executing Recovery Plan
Component Recovering
Reactivate [Active Transactions Count == 0]]/Notify Active

Figure 3.3 Integrated adaptation and recovery state machine executed by Connector Control

3.1.6 Service Recovery

While the service is in the recovering state, Connector Control must not forward any requests and must ensure that all failed transactions are restarted when the service is recovered.

Recovering a service from failure is handled by the connector using the MAPE-K loop model for self-healing and self-configuration, as explained next. The monitoring activity of MAPE-K notifies the RAC of the service failure. When Connector Control receives a failure notification, it notifies the Service Request Coordinator of the failure and then transitions to the Analyzing Failure Events state (Figure 3.3).

The Analyzing Failure Events state corresponds to the analysis activity of MAPE-K where the RAC identifies all transactions that were interrupted due to service failure. The RAC determines that a transaction has failed if either the SAQ or SRQ contain a request that initiates a transaction with the service but neither the RFQ nor the RRQ contains a response that completes that transaction. When failure analysis is completed, Connector Control transitions to the Planning for Recovery state.

The Planning for Recovery state corresponds to the planning activity of MAPE-K where the RAC determines the recovery plan for the failed transactions. The plan identifies which requests must be resent to the recovered service so that failed transactions are restarted at the recovered service. The recovery plan is determined by executing the following recovery policy:

- First, the RAC forwards previous requests of every failed dialog that the service was engaged in before it failed. These requests are recovered from the SRQ and are forwarded sequentially in the same order they were processed before service failure to ensure that the recovered service also processes these requests in that order.
- Second, the RAC forwards the requests of failed transactions queued in the SAQ, which contains pending requests that were lost by the failed service before the RAC received the responses to these requests. Note that at this step, if a request

that is being forwarded is of a dialog, then (from the previous step) the service must have already received all previous requests of this dialog.

- Third, the RAC forwards all requests in the SPQ, which are new requests that have been received while the service is in the recovering state, to the recovered service.

The Executing Recovery Plan state corresponds to the execution activity of MAPE-K where the RAC restores all requests that must be resent to the recovered service by moving these requests from the SRQ and SAQ to the SPQ, as specified in the recovery plan. When all requests are restored, Connector Control transitions to the Component Recovering state in which the connector waits until the service is relocated and instantiated by the Recovery and Adaptation Manager (this manager will be discussed in detail in chapter 5), and then has its connection with the recovered service established. Eventually, when Connector Control receives the Reactive command, Connector Control transitions to the Active state and notifies the Service Request Coordinator that the service is active so that the Service Request Coordinator resumes sending requests queued in the SPQ to Connector Control.

3.1.7 Service Request Coordinator State Machine

Based on the discussion in the previous section, the Service Request Coordinator must forward to Connector Control certain types of client requests based on the current state of the service, as shown in Figure 3.4. While the service is active (Figure 3.4), the Service Request Coordinator forwards all client requests it receives to Connector Control and also queues these requests in the SPQ.

When the Service Request Coordinator is notified that the service is passivating, it transitions to the Passivating state. The behavior of the Service Requests Coordinator while in this state is similar to its behavior in the Active state with one exception: in the Passivating state, the Service Request Coordinator does not forward to Connector Control any requests that initiate a new transaction with the service, and instead, queues such requests in the SPQ. Eventually, the Service Request Coordinator is notified that the service has become quiescent, causing the Service Request Coordinator to transition to the Quiescent state. While in the Quiescent state, the Service Request Coordinator does not forward any requests to Connector Control and instead queues them in the SPQ. Finally, when service adaptation is completed, the Service Request Coordinator receives a notification that the service is active, causing the Service Request Coordinator to transition to the Active state and to forward all requests queued in the SPQ to Connector Control.

When service failures occur, the Service Request Coordinator transitions to the Failed state. While in the Failed state, the Service Request Coordinator holds all client requests it receives in the SPQ. The Service Request Coordinator may also receive messages from the execution activity of MAPE-K to restore any client requests that were lost due to service failure. As a result, the Service Request Coordinator moves these requests from the SRQ and the SAQ to the head of the SPQ so that these requests are resent to the recovered service. Finally, when the service is recovered, the Service Request Coordinator forwards all requests stored in the SPQ and then transitions back to the Active state.

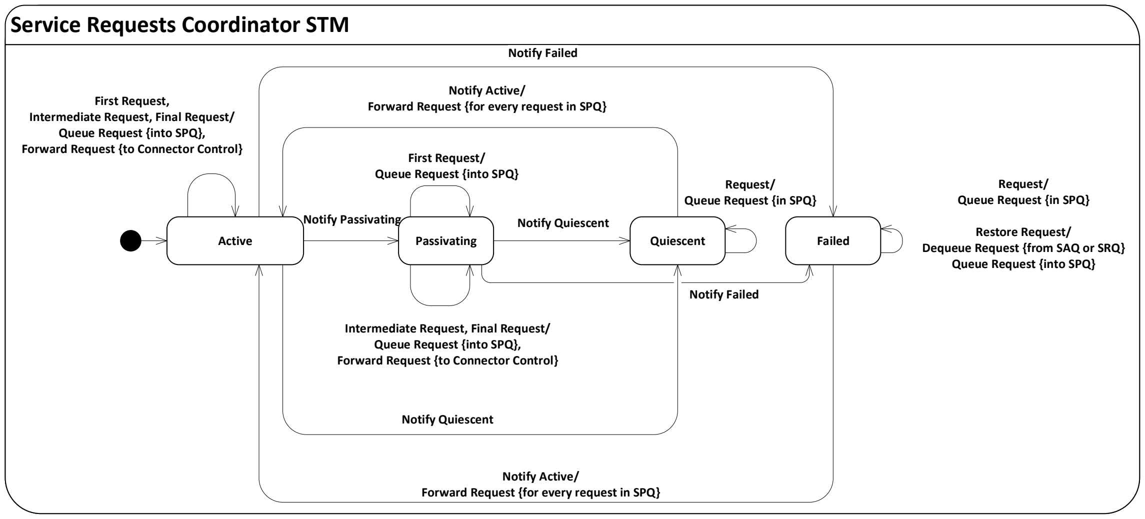

Service Requests Coordinator STM
Notify Failed
First Request,
Intermediate Request, Final Request/
Queue Request {into SPQ},
Forward Request {to Connector Control}
Notify Active/
Forward Request {for every request in SPQ}
First Request/
Queue Request {into SPQ}
Request/
Queue Request {in SPQ}
Request/
Queue Request {in SPQ}
Restore Request/
Dequeue Request {from SAQ or SRQ}
Queue Request {into SPQ}
Notify Passivating
Notify Quiescent
Active
Passivating
Quiescent
Failed
Notify Failed
Intermediate Request, Final Request/
Queue Request {into SPQ},
Forward Request {to Connector Control}
Notify Quiescent
Notify Active/
Forward Request {for every request in SPQ}

Figure 3.4 State machine executed by Service Request Coordinator

3.2 Handling Non-Idempotent Operations

This section discusses extending the design of the RAC in section 3.1 to handle recovery and adaptation of *stateful* services with both idempotent and non-idempotent operations. It is assumed that the state of the stateful service is maintained by a transactional processing system that supports committing, aborting, and preparing transactions (Bernstein and Newcomer, 2009). The transactional processing system handles recovery of the service's state to a consistent state by using a transactional log to:

- Undo all transactions that have either been aborted or did not complete before service failure.
- Redo transactions that have been committed before service failure.

- Restore the state of prepared transactions until these transactions are either committed or aborted.

Since the service is a stateful component with non-idempotent operations, then the RAC must ensure that (1) committing the client's transaction at the service side and (2) updating the queues at the Service Request Manager are performed as an atomic operation. To achieve this behavior, the RAC forwards each client request to the service by initiating a transaction using the Two-Phase Commit (2PC) protocol (Bernstein and Newcomer, 2009). In this approach, Connector Control of the RAC acts as the coordinator of the 2PC transaction while the service and the Service Request Coordinator act as participants of this transaction, as explained next.

During normal execution, when there are no failures, the interaction between the RAC and the service is as follows (Figure 3.5):

1. When Connector Control of the service RAC (not shown in Figure 3.5) receives a client request, it forwards the client request to the service in a 2PC transaction. This request corresponds to the Prepare To Commit message in the 2PC protocol. Connector Control also forwards this request to the Service Request Coordinator, which is a second participant of this 2PC transaction. As a result, the Service Request Coordinator prepares to commit the client request by moving this request from the Service Pending Queue to the Service Active Queue and then acknowledges preparing the transaction to Connector Control.

2. The service prepares to commit the client request and then sends the response to the RAC. The service response corresponds to the Ready To Commit message in the 2PC protocol.
3. Connector Control then sends the Commit message to both the service and the Service Request Coordinator. The Service Request Coordinator then commits the transaction by moving the client request from the Service Active Queue to the Service Recovery Queue and then acknowledges committing the transaction to Connector Control.
4. The service commits the prepared transaction and then sends the Committed message to the RAC which completes this 2PC transaction.
5. The RAC forwards the service response to the requesting client.

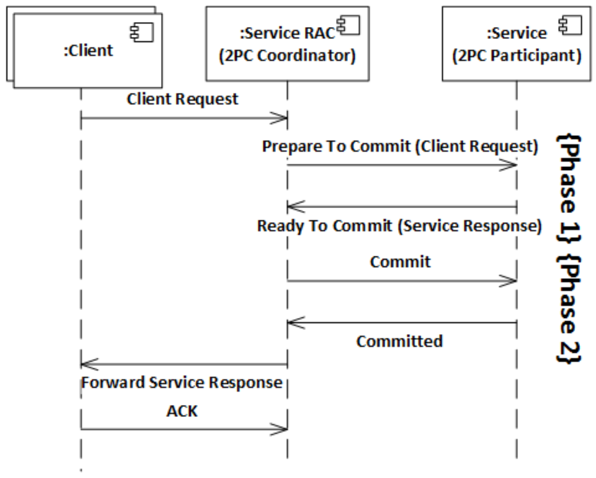


Figure 3.5 Handling stateful services using two-phase commit

To satisfy this behavior, Connector Control (CC) executes the state machine shown in Figure 3.6. In this STM, CC is initially in the Waiting for Request state (Figure 3.6). During this state, If CC receives a client request, then there are two cases to consider: whether this request initiates a dialog with the service or not. If the request initiates a dialog, then the actions are to (1) increment the transaction count, (2) forward the request to the service, and then (3) transition to the Processing state. On the other hand, if the request is of a single request/response transaction, then the actions are to (1) increment the transaction count, (2) request the service to prepare to commit this request since this is the only request in this transaction, and then (3) transition to the Processing state.

While in the Processing state, CC reacts to the various events as follows:

- If CC receives a request that initiates a new transaction with the service, then CC (1) increments the transaction count and (2) either forwards the request to the service (if this request initiates a new dialog as determined by the message header) or requests the service to prepare to commit this request (if the message header indicates that the transaction comprises a single request/response messages).
- If CC receives an intermediate request of a dialog, then CC forwards this request to the service.
- If CC receives a final request of a dialog, then CC requests the service to prepare to commit this transaction, since this is the last request of this transaction.

- If the event is the first response or is an intermediate response of a dialog, then CC forwards the response to the requesting client.
- If CC receives a Read Only response from the service (Bernstein and Newcomer, 2009), then this transaction is a read-only transaction that does not update the service’s state. As a result, the actions are to decrement the transaction count and forward the service response to the requesting client. If this is the last transaction that that service is currently engaged in, then CC transitions to the Waiting for Request state.
- If CC receives a Ready To Commit response, the action is to send the Commit request to the service.
- If CC receives a Refuse To Commit response, the action is to send the Abort message to the service.
- If CC receives a Committed response, the action is to forward the service response, which was held by the RAC until the completion of this transaction, to the requesting client. This event also causes CC to decrement the transaction count. If this is the last transaction that the service is engaged in, then CC transitions to the Waiting for Request state.
- If CC receives an Aborted response, the action is to forward the service response, which was held by the RAC until the completion of this transaction, to the requesting client. This event also causes CC to decrement the transaction count. If this is the last transaction that the service is engaged in, then CC transitions to the Waiting for Request state.

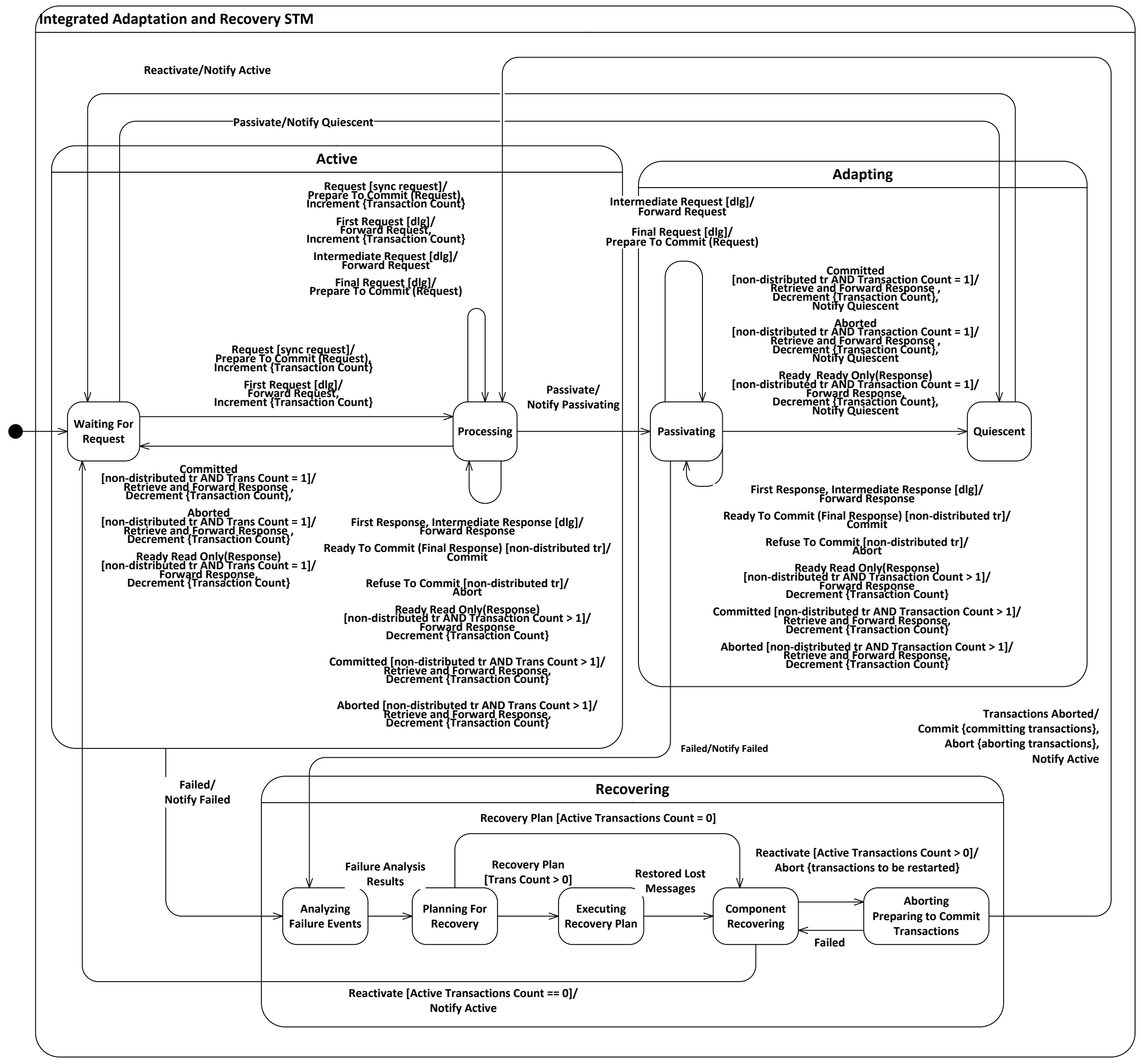

Integrated Adaptation and Recovery STM
Reactivate/Notify Active
Passivate/Notify Quiescent
Active
Request [sync request]/ Prepare To Commit (Request), Increment {Transaction Count}
First Request [dlg]/ Forward Request, Increment {Transaction Count}
Intermediate Request [dlg]/ Forward Request
Final Request [dlg]/ Prepare To Commit (Request)
Request [sync request]/ Prepare To Commit (Request), Increment {Transaction Count}
First Request [dlg]/ Forward Request, Increment {Transaction Count}
Waiting For Request
Processing
Passivate/ Notify Passivating
Committed [non-distributed tr AND Trans Count = 1]/ Retrieve and Forward Response , Decrement {Transaction Count},
Aborted [non-distributed tr AND Trans Count = 1]/ Retrieve and Forward Response , Decrement {Transaction Count}
Ready Read Only(Response) [non-distributed tr AND Trans Count = 1]/ Forward Response, Decrement {Transaction Count}
First Response, Intermediate Response [dlg]/ Forward Response
Ready To Commit (Final Response) [non-distributed tr]/ Commit
Refuse To Commit [non-distributed tr]/ Abort
Ready Read Only(Response) [non-distributed tr AND Transaction Count > 1]/ Forward Response Decrement {Transaction Count}
Committed [non-distributed tr AND Trans Count > 1]/ Retrieve and Forward Response, Decrement {Transaction Count}
Aborted [non-distributed tr AND Trans Count > 1]/ Retrieve and Forward Response, Decrement {Transaction Count}
Adapting
Intermediate Request [dlg]/ Forward Request
Final Request [dlg]/ Prepare To Commit (Request)
Committed [non-distributed tr AND Transaction Count = 1]/ Retrieve and Forward Response , Decrement {Transaction Count}, Notify Quiescent
Aborted [non-distributed tr AND Transaction Count = 1]/ Retrieve and Forward Response , Decrement {Transaction Count}, Notify Quiescent
Ready Ready Only(Response) [non-distributed tr AND Transaction Count = 1]/ Forward Response, Decrement {Transaction Count}, Notify Quiescent
Passivating
Quiescent
First Response, Intermediate Response [dlg]/ Forward Response
Ready To Commit (Final Response) [non-distributed tr]/ Commit
Refuse To Commit [non-distributed tr]/ Abort
Ready Read Only(Response) [non-distributed tr AND Transaction Count > 1]/ Forward Response Decrement {Transaction Count}
Committed [non-distributed tr AND Transaction Count > 1]/ Retrieve and Forward Response, Decrement {Transaction Count}
Aborted [non-distributed tr AND Transaction Count > 1]/ Retrieve and Forward Response, Decrement {Transaction Count}
Transactions Aborted/ Commit {committing transactions}, Abort {aborting transactions}, Notify Active
Failed/Notify Failed
Failed/ Notify Failed
Recovering
Recovery Plan [Active Transactions Count = 0]
Failure Analysis Results
Recovery Plan [Trans Count > 0]
Restored Lost Messages
Reactivate [Active Transactions Count > 0]/ Abort {transactions to be restarted}
Analyzing Failure Events
Planning For Recovery
Executing Recovery Plan
Component Recovering
Aborting Preparing to Commit Transactions
Failed
Reactivate [Active Transactions Count == 0]/ Notify Active

Figure 3.6 State machine executed by Connector Control for handling services with non-idempotent operations

In this pattern, the service can be dynamically removed or replaced after it has completed all 2PC transactions that it is currently engaged in. In case of a service failure, when the RAC is notified of the service failure, the RAC determines a recovery action for each 2PC transaction it has initiated with the service as follows:

1. If the RAC has forwarded a client request to the service but the service failure occurred during phase 1 of the 2PC transaction, the RAC recovers the client request by moving it from the Service Active Queue to the Service Pending Queue. When the service is recovered, the RAC instructs the recovered service to abort this transaction so that the recovered service aborts the transaction if it has previously been prepared to commit. Note that if the service failed before preparing to commit the transaction, it ignores the Abort message from the RAC. Since the client request is saved in the Service Pending Queue, then the RAC eventually restarts this transaction with the recovered service.
2. If the RAC has received a service response from the service but has not yet forwarded the Commit message to the service (i.e., service failure occurred before initiating phase 2 of the 2PC transaction), then the service must have prepared to commit this transaction before it failed. As a result, the RAC sends the Commit message to the service after it has recovered so that it commits this transaction. When the recovered service commits the transaction, it sends the Committed message to RAC which completes this transaction.
3. If the RAC has forwarded the Commit message to the service but the service failure occurred during phase 2 of the 2PC transaction, the RAC resends the Commit message to the service after it has recovered. As a result, the recovered service commits the prepared transaction and sends the Committed message to the RAC. Note that the Commit message itself is idempotent. That is, if the service has committed the transaction before failure, then receiving a duplicate Commit

message causes the recovered service to send the Committed message to the RAC.

3.3 Recovery and Adaptation Patterns in SOA Patterns

This section describes how the RAC design discussed in the previous section can be used to handle adaptation and recovery of components in other more complex architectural patterns (Gomaa, 2011).

3.3.1 Asynchronous Message Communication with Callback Pattern

Typical client/service communication uses the Synchronous Message Communication with Reply pattern, in which the client sends a message to the service and waits for a response. In the Asynchronous Message Communication with Callback pattern (Figure 3.7), a client sends an asynchronous request to the service but can continue executing and receive the service response later. The request sent by the client contains a callback handle that the service uses when it finishes processing the client request so that it can send the response back to the client. A client in this pattern does not send another request to the service until it receives a response to the previous request.

Since in this pattern a client sends one request at a time to the service, the RAC (shown in Figure 3.1) handles requests and responses for this pattern in the same way as for synchronous communication with reply. Thus, although the client behaviour is different, the service behaviour is not. For this reason, the adaptation and recovery for the Asynchronous Message Communication with Callback pattern is handled in the same way as described in sections 3.1 and 3.2.

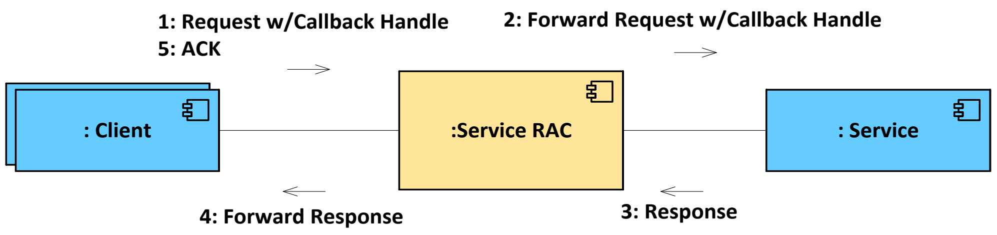


Figure 3.7 Asynchronous message communication with callback handle pattern

3.3.2 Service Registration Pattern

In service-oriented architectures, a service registers its name, location and service description with a broker, which acts as an intermediary between the clients and the service. In the Service Registration pattern (Figure 3.8), the service initiates a transaction with the broker by sending it a registration request containing the service information. The broker then registers the service and sends an acknowledgement to the service. The service can also re-register with the broker if it moves its location, which requires another transaction between the service and the broker.

From the adaptation and recovery point of view, this pattern can be treated as a client that communicates with a service using the Synchronous Message Communication with Reply pattern. Thus, the adaptation and recovery patterns for this architectural pattern are exactly the same as described in sections 3.1 and 3.2.

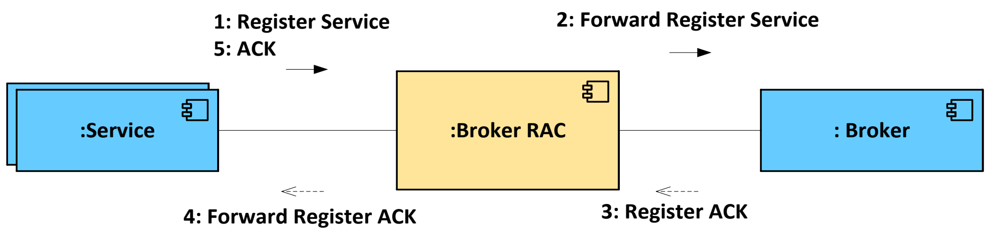


Figure 3.8 Service registration pattern

3.3.3 Broker Handle Pattern

After the service has registered with the broker, clients use the broker to locate the service. In the Broker Handle pattern (Figure 3.9), a client sends a request to the broker to obtain the service's handle. The broker then sends a response to the client containing the service's handle as a parameter. The client then uses the service's handle to interact with the service.

In this pattern, a client initiates two sequential transactions by first initiating a transaction with the broker to obtain the service's handle and then by initiating a transaction with the service using the service's callback handle. As a result, these transactions can fail and be recovered independently of each other.

A broker is adapted after it has completed all the requests it has received, including brokering requests from clients requesting a handle and service requests for registration. New requests are held up until the broker has been relocated. In the case of a broker failure, all requests it is dealing with are aborted and only restarted when the broker has been relocated and instantiated. Both adaptation and recovery are carried out as described in Sections 3.1 and 3.2.

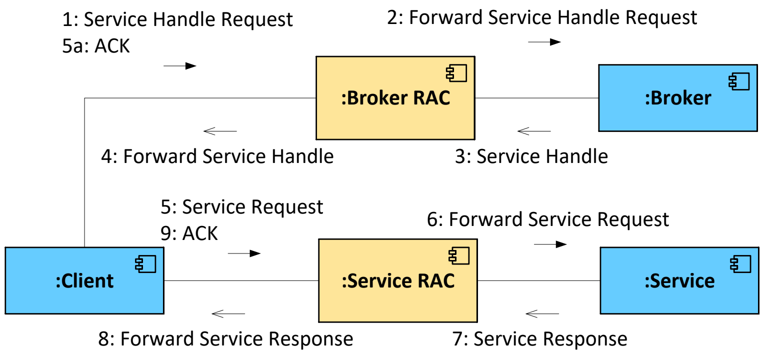

1: Service Handle Request
5a: ACK
2: Forward Service Handle Request
:Broker RAC
:Broker
4: Forward Service Handle
3: Service Handle
5: Service Request
9: ACK
6: Forward Service Request
:Client
:Service RAC
:Service
8: Forward Service Response
7: Service Response

Figure 3.9 Broker handle pattern

3.3.4 Service-Oriented Architectures

In service-oriented architectures (SOAs), the goal is to increase loose coupling between services so that instead of services depending on each other, coordinators are provided for situations where multiple services need to be accessed, and access to them needs to be coordinated and/or sequenced (see Figure 3.10). We consider that the coordinator may interact with the services sequentially and/or concurrently and that the interaction between the coordinator and the multiple services involves a compound transaction that can be broken down into an atomic, independent transaction between the coordinator and each service, as described in the next subsection.

In this pattern, when any of the services fail, the service's RAC restarts each failed transaction with the service without affecting other transactions that the coordinator is currently engaged in with other services. Thus, the recovery and adaptation patterns for services in this pattern are exactly the same as discussed in sections 3.1 and 3.2. The remainder of this section describes recovery and adaptation of the coordinator.

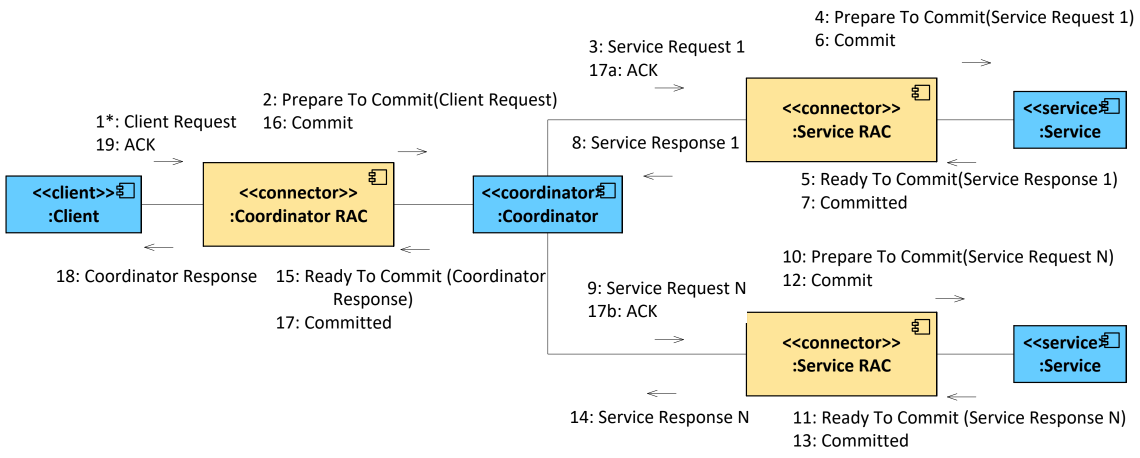


Figure 3.10 SOA architectural pattern

For coordinators, we assume the general case in which the coordinator is a stateful component. Therefore, the Coordinator RAC must forward client requests to the Coordinator in 2PC transactions using the same approach described section 3.2 so that updating the Coordinator RAC’s queues and updating the Coordinator’s internal state is an atomic operation. Therefore, the Coordinator RAC coordinates the 2PC transaction it initiates with the Coordinator while the Coordinator coordinates access to the services. The behavior of the Coordinator RAC and the Coordinator is as follows:

1. When the Coordinator RAC receives a client request (message 1 (m1) in Figure 3.10), it forwards this client request to the coordinator in a 2PC transaction (m2). This message corresponds to the Prepare to Commit message in the 2PC protocol.

2. When the Coordinator receives the client request, it initiates a compound transaction, which consists of initiating a constituent atomic transaction with each service.
3. When the Coordinator receives responses from all services (m8 and m14), it prepares to commit the compound transaction it has initiated in the previous step and then sends its response (m15) to the Coordinator RAC. This response corresponds to the Ready To Commit message for the 2PC transaction initiated in step 1.
4. The Coordinator RAC then sends the Commit message (m16) to the Coordinator.
5. The Coordinator then commits the previously prepared compound transaction, sends ACK messages to the service RACs so that these connectors can safely remove messages for this transaction from their queues, and then sends Committed (m17) to the Coordinator RAC. At this point, the 2PC transaction between the Coordinator's RAC and the Coordinator is completed.
6. The Coordinator's RAC sends the Coordinator's response to the client (m18).

Adaptation and Recovery of Coordinators.

In the case of a client interacting with a coordinator, if the coordinator needs to be adapted, then the client request needs to be completed before adaptation. This means that the entire 2PC transaction between the Coordinator RAC and the Coordinator must complete before adaptation can take place, since completion of this 2PC transaction ensures that the last compound transaction initiated by the coordinator is also completed.

In the case of coordinator failure, when the coordinator is recovered, the recovered coordinator must abort the last compound transaction it initiated, if this compound transaction has not been prepared to commit before failure. Since the interaction between the coordinator's RAC and the coordinator involves a 2PC transaction, then the coordinator's RAC executes similar recovery actions to those described in section 3.2 to recover this 2PC transaction in case it failed, as follows:

1. If the Coordinator RAC has forwarded a client request to the Coordinator but the coordinator failure occurred during phase 1 of the 2PC transaction, the coordinator RAC saves the client request by moving it from the Coordinator Active Queue to the Coordinator Pending Queue. When the coordinator is recovered, the coordinator RAC instructs the recovered coordinator to abort this transaction so that the recovered coordinator aborts the compound transaction, even if it has been prepared to commit. Since the client request is stored in the Coordinator Pending Queue, then eventually the coordinator's RAC restarts this transaction with the recovered coordinator. Since transactions to coordinators can be restarted, a recovered coordinator may send duplicate requests to Service RACs. These Service RACs detect and discard duplicate requests by comparing message sequence numbers of incoming messages with previously received messages. Furthermore, if responses of duplicate requests are queued in the Response Recovery Queue (RRQ), then these service RACs resend these responses to the recovered coordinator. Note that from Figure 3.10, a service RAC does not discard a service response for any transaction until it receives an

ACK message from the coordinator that initiated this transaction. Because a coordinator sends ACK messages to service RACs only after it has committed the compound transaction it initiated, this ensures that a service RAC can always recover responses of the duplicate requests it receives from recovered coordinators. Note that if a Service RAC does not maintain in its RRQ the response of a duplicate request, the service RAC forwards the response to the coordinator after it receives this response from the service.

2. If the coordinator RAC has received a ready to commit response from the coordinator but has not yet sent the Commit message to the coordinator (i.e., a coordinator failure occurred before initiating phase 2 of the 2PC transaction), the coordinator must have prepared to commit this transaction before it failed. As a result, the coordinator's RAC sends the Commit message to the recovered coordinator for this transaction so that it commits this transaction. When the recovered coordinator commits the transaction, it sends the Committed message to the coordinator RAC, which completes this transaction.
3. If the coordinator RAC has forwarded the Commit message to the coordinator but has not yet received the Committed message (i.e., a coordinator failure occurred during phase 2 of the 2PC transaction), the coordinator's RAC resends the Commit message to the recovered coordinator. As a result, the recovered coordinator commits the prepared transaction, sends ACK messages to the service RACs, and then sends the Committed message to the coordinator's RAC.

3.3.5 Distributed Transaction Pattern

This section considers the recovery and adaptation patterns for the distributed transaction pattern in which there is a requirement that an atomic (i.e. indivisible) distributed transaction involving updates at multiple services must be either committed by all the services (see Figure 3.11) or aborted by all the services (see Figure 3.12). In this pattern, we consider the general case in which coordinators and services are both stateful components. Therefore, there is a 2PC transaction between each coordinator RAC and its coordinator and each service RAC and its service. Since the coordinators in this pattern also initiate 2PC transactions with the multiple services, then this result in a tree of 2PC transactions (Vossen and Weikum, 2001). The tree of the 2PC transactions is needed to ensure that the 2PC transactions between the coordinator RAC, the coordinator, the service RACs, and the services are atomic (i.e. indivisible) such that either all 2PC transactions in this tree are committed or none of them are. In this tree of 2PC transactions, the overall decision as to whether to commit or abort this tree is controlled by the coordinator RAC since this RAC is the root of this tree, as shown in Figure 3.13.

In this tree of 2PC transactions:

1. The coordinator RAC coordinates the 2PC transaction it initiates with the coordinator.
2. The coordinator (1) participates in the 2PC transaction initiated by the coordinator RAC and (2) coordinates the distributed transaction it initiates with the multiple services via the service RACs.
3. A service RAC (1) participates in the 2PC transaction initiated by the coordinator and (2) coordinates the distributed transaction it initiates with it service.

Since service RACs are participants of the transactions initiated by coordinators, then a service RAC does not commit the transaction it initiates with its service until it has received the coordinator's decision on whether to commit or abort this transaction. Furthermore, since coordinators are participants of the transactions initiated by coordinator RACs, then a coordinator does not commit the distributed transaction it initiates with the multiple services until it has received the coordinator RAC's decision on whether to commit or abort this transaction.

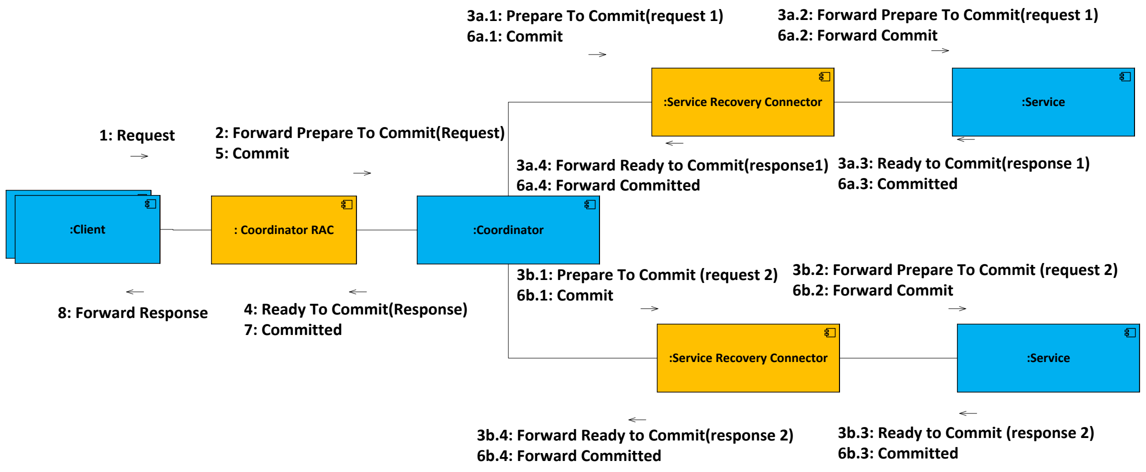

3a.1: Prepare To Commit(request 1)
6a.1: Commit
3a.2: Forward Prepare To Commit(request 1)
6a.2: Forward Commit
:Service Recovery Connector
:Service
1: Request
2: Forward Prepare To Commit(Request)
5: Commit
3a.4: Forward Ready to Commit(response1)
6a.4: Forward Committed
3a.3: Ready to Commit(response 1)
6a.3: Committed
:Client
: Coordinator RAC
:Coordinator
3b.1: Prepare To Commit (request 2)
6b.1: Commit
3b.2: Forward Prepare To Commit (request 2)
6b.2: Forward Commit
8: Forward Response
4: Ready To Commit(Response)
7: Committed
:Service Recovery Connector
:Service
3b.4: Forward Ready to Commit(response 2)
6b.4: Forward Committed
3b.3: Ready to Commit (response 2)
6b.3: Committed

Figure 3.11 Two phase commit pattern – transaction commit case

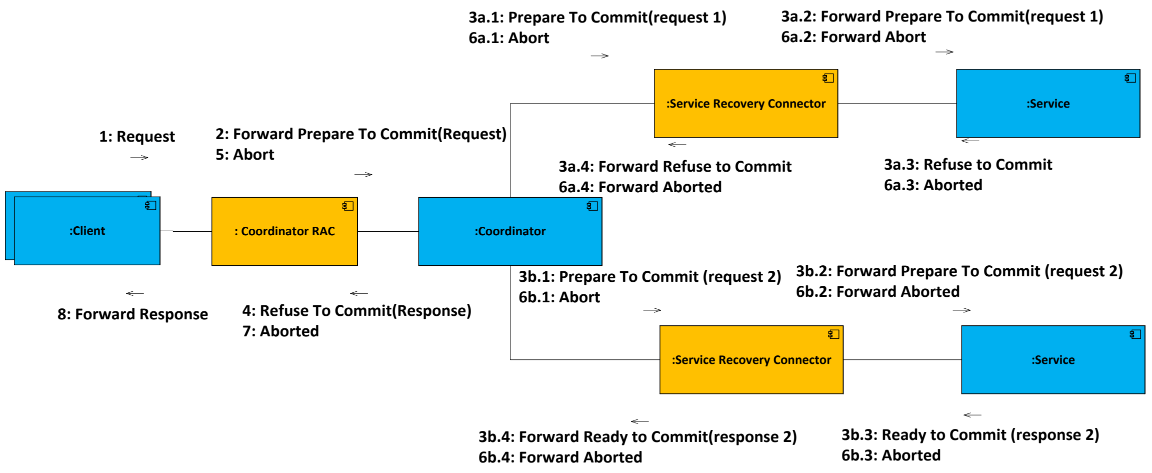

3a.1: Prepare To Commit(request 1)
6a.1: Abort
3a.2: Forward Prepare To Commit(request 1)
6a.2: Forward Abort
:Service Recovery Connector
:Service
1: Request
2: Forward Prepare To Commit(Request)
5: Abort
3a.4: Forward Refuse to Commit
6a.4: Forward Aborted
3a.3: Refuse to Commit
6a.3: Aborted
:Client
: Coordinator RAC
:Coordinator
3b.1: Prepare To Commit (request 2)
6b.1: Abort
3b.2: Forward Prepare To Commit (request 2)
6b.2: Forward Aborted
8: Forward Response
4: Refuse To Commit(Response)
7: Aborted
:Service Recovery Connector
:Service
3b.4: Forward Ready to Commit(response 2)
6b.4: Forward Aborted
3b.3: Ready to Commit (response 2)
6b.3: Aborted

Figure 3.12 Two phase commit pattern – transaction abort case

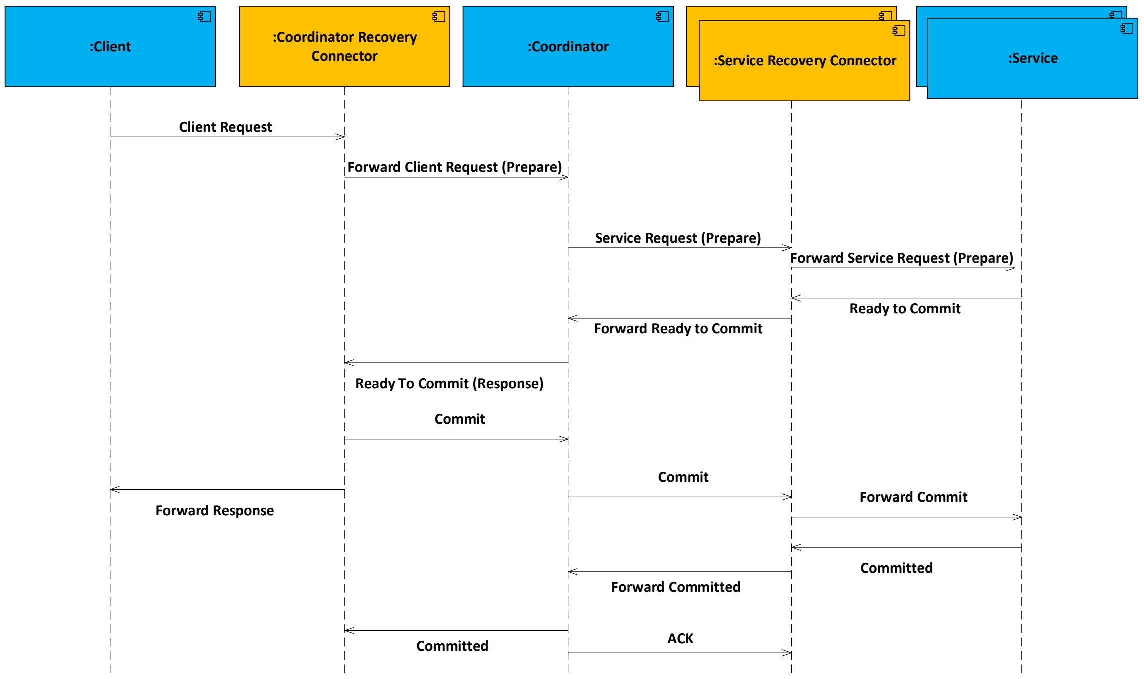

:Client
:Coordinator Recovery Connector
:Coordinator
:Service Recovery Connector
:Service
Client Request
Forward Client Request (Prepare)
Service Request (Prepare)
Forward Service Request (Prepare)
Ready to Commit
Forward Ready to Commit
Ready To Commit (Response)
Commit
Commit
Forward Commit
Forward Response
Committed
Forward Committed
ACK
Committed

Figure 3.13 Tree of 2PC transactions

<u>Recovery and Adaptation of Coordinators</u>

In this pattern, the interaction between the coordinator RAC and the coordinator is exactly the same as the interaction between the coordinator RAC and the coordinator in the SOA pattern (see section 3.3.4). Therefore, the recovery and adaptation pattern for coordinators is exactly the same as the one described in section 3.3.4.

Recovery of Services

Unlike the SOA pattern in which a service RAC can determine independently the decision of the 2PC transactions it initiates with its service (see section 3.3.4), the service RACs involved in the Distributed Transaction pattern act as participants to the distributed transactions initiated by the transaction coordinators, and therefore these RACs must ensure that the decision message they send to their services must always agree with the decision determined by these coordinators so that the atomicity of the entire distributed transaction that involves the multiple services is preserved.

In order for the service RAC to satisfy this requirement, Connector Control (CC) executes the state machine shown in Figure 3.14. In this STM, CC is initially in the Waiting for Request state. During this state, If CC receives a Prepare To Commit request from the distributed transaction coordinator, then the actions are to (1) increment the transaction count, (2) forward the Prepare To Commit message to the service, and then (3) transition to the Processing state.

While in the Processing state, CC reacts to the various events as follows:

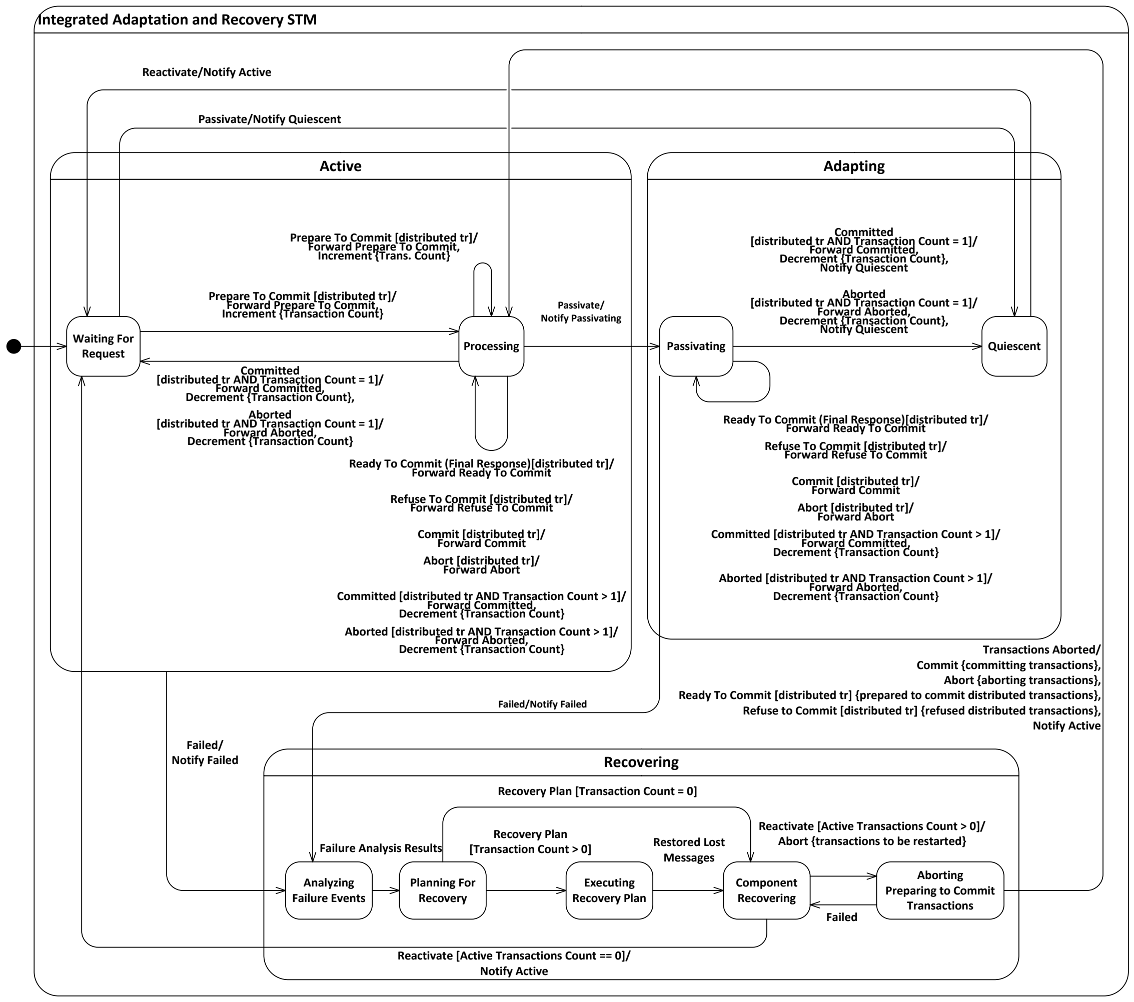
- If CC receives a Prepare To Commit request, then CC (1) increments the active transaction count and (2) forwards the Prepare To Commit message to the service.

- If CC receives a Ready To Commit response, then this RAC must not make a decision for this transaction and must instead wait for this decision from the transaction coordinator. Therefore, the action is to forward Ready To Commit to the transaction coordinator.
- If CC receives a Refuse To Commit response, then this service is unable to commit this transaction. Thus, the action is to forward Refuse To Commit to the transaction coordinator.
- If CC receives a Commit request for a distributed transaction, then the action is to forward the Commit request to the service.
- If CC receives an Abort request for a distributed transaction, then the action is to forward the Abort request to the service.
- If CC receives a Committed response, then the action is to forward the Committed response to the transaction coordinator. This event also causes CC to decrement the transaction count. If this counter indicates that the service is not engaged in any other transactions, then CC transitions to the Waiting for Request state.
- If CC receives an Aborted response, then the action is to forward the Aborted response to the transaction coordinator. This event also causes CC to decrement the transaction count. If this counter indicates that the service is not engaged in any other transactions, then CC transitions to the Waiting for Request state.

In this pattern, the service can be dynamically removed or replaced after it has completed all distributed transactions that it is currently engaged in. In case of a service failure, when the service RAC is notified of service failure, the service RAC determines a recovery action for each distributed transaction as follows:

- If the service RAC has forwarded the Prepare To Commit to the service but the service failure occurred during phase 1 of the 2PC transaction, the RAC moves this transaction from the Service Active Queue to the Service Pending Queue. When the service is recovered, to avoid sending duplicate requests to the recovered service, the RAC instructs the recovered service to abort this transaction. Note that if the service failed before preparing to commit the transaction, it ignores the Abort message from the RAC. Since the transaction is saved in the Service Pending Queue, then the RAC will eventually restart this transaction with the recovered service.
- If the RAC has received Ready To Commit from the service but has not yet forwarded the decision message (i.e. Commit or Abort messages) to the service, then the action is to send the Ready to Commit message to the transaction coordinator.
- If the RAC has received Refuse To Commit from the service but has not yet forwarded the Abort message to the service, then the action is to send the Refuse to Commit message to the transaction coordinator.
- If the RAC has forwarded either the Commit or Abort message to the service but the service failure occurred during phase 2 of the 2PC transaction, the RAC resends the Commit or Abort message to the service after it has recovered. As a

result, the recovered service will either commit or aborts the transaction and then send the Committed or Aborted message to the RAC.



Integrated Adaptation and Recovery STM
Reactivate/Notify Active
Passivate/Notify Quiescent
Active
Adapting
Prepare To Commit [distributed tr]/ Forward Prepare To Commit, Increment {Trans. Count}
Prepare To Commit [distributed tr]/ Forward Prepare To Commit, Increment {Transaction Count}
Passivate/ Notify Passivating
Committed [distributed tr AND Transaction Count = 1]/ Forward Committed, Decrement {Transaction Count}, Notify Quiescent
Aborted [distributed tr AND Transaction Count = 1]/ Forward Aborted, Decrement {Transaction Count}, Notify Quiescent
Waiting For Request
Processing
Passivating
Quiescent
Committed [distributed tr AND Transaction Count = 1]/ Forward Committed, Decrement {Transaction Count},
Aborted [distributed tr AND Transaction Count = 1]/ Forward Aborted, Decrement {Transaction Count}
Ready To Commit (Final Response)[distributed tr]/ Forward Ready To Commit
Refuse To Commit [distributed tr]/ Forward Refuse To Commit
Commit [distributed tr]/ Forward Commit
Abort [distributed tr]/ Forward Abort
Committed [distributed tr AND Transaction Count > 1]/ Forward Committed, Decrement {Transaction Count}
Aborted [distributed tr AND Transaction Count > 1]/ Forward Aborted, Decrement {Transaction Count}
Ready To Commit (Final Response)[distributed tr]/ Forward Ready To Commit
Refuse To Commit [distributed tr]/ Forward Refuse To Commit
Commit [distributed tr]/ Forward Commit
Abort [distributed tr]/ Forward Abort
Committed [distributed tr AND Transaction Count > 1]/ Forward Committed, Decrement {Transaction Count}
Aborted [distributed tr AND Transaction Count > 1]/ Forward Aborted, Decrement {Transaction Count}
Transactions Aborted/ Commit {committing transactions}, Abort {aborting transactions}, Ready To Commit [distributed tr] {prepared to commit distributed transactions}, Refuse to Commit [distributed tr] {refused distributed transactions}, Notify Active
Failed/Notify Failed
Failed/ Notify Failed
Recovering
Recovery Plan [Transaction Count = 0]
Reactivate [Active Transactions Count > 0]/ Abort {transactions to be restarted}
Failure Analysis Results
Recovery Plan [Transaction Count > 0]
Restored Lost Messages
Analyzing Failure Events
Planning For Recovery
Executing Recovery Plan
Component Recovering
Aborting Preparing to Commit Transactions
Failed
Reactivate [Active Transactions Count == 0]/ Notify Active

Figure 3.14: State machine executed by Connector Control for handling distributed transactions

4 DESIGN OF RECOVERY AND ADAPTATION CONNECTORS IN ASYNCHRONOUS ARCHITECTURAL PATTERNS

The previous chapter discussed the design of the service RAC that receives both input requests to a service and output responses from the service for several SOA-related patterns. This chapter discusses the design of a different type of the RAC that is used in various asynchronous architectural patterns. Unlike the service RAC, a RAC in an asynchronous pattern does not receive output responses from its component, and as a result, handles only input messages to its component.

This chapter is organized is follows. Section 4.1 discusses the design of the consumer RAC for the unidirectional asynchronous message communication pattern when consumers are stateless. Section 4.2 shows how the approach can be extended to handle state-dependent consumers. Section 4.3 discusses how the same RAC is also applicable for other asynchronous patterns including the bidirectional asynchronous message communication pattern, the subscription/notification communication pattern, the master/slave architectural pattern, and various control patterns.

4.1 Design of the Consumer RAC in the Unidirectional Asynchronous Message Communication

In the unidirectional asynchronous message communication (Figure 4.1), one or more producers send one or more asynchronous messages to the consumer. The messages from a producer to the consumer do not require any responses from the consumer to the producer. As a result, producers continue execution immediately after sending messages to the consumer. It is assumed that no dialogs are involved between producers and the consumer, since the consumer does not send any responses to producers. Furthermore, it is assumed that each message from the producer initiates a new transaction with the consumer. A transaction in this pattern consists of (1) the producer sending an asynchronous message to the consumer, (2) the consumer consuming the producer's message, and (3) the consumer sending an acknowledgement message back to the consumer RAC. The remainder of this section discusses the design of the consumer RAC and how it handles adaptation and recovery concerns of the consumer with the assumption that the consumer is a stateless component. Section 4.2 discusses handling state-dependent consumers.

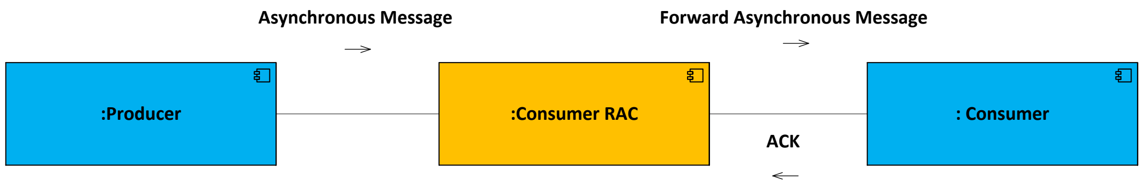


Figure 4.1 Unidirectional asynchronous message communication between a producer and a consumer

4.1.1 Design of the Consumer RAC

If the consumer is a stateless component, then the message sequencing between the consumer RAC and the consumer is as follows (Figure 4.1):

1. When the consumer RAC receives a message from the producer, it forwards the message to the consumer.

2. When the consumer is finished with the producer’s message, it sends the ACK message to the consumer RAC. This message serves as an acknowledgement to the consumer RAC that the producer’s message is not needed anymore and thus can be removed from the connector’s queues. Note that although it is not necessary for asynchronous communication, the ACK message is needed for recovery and adaptation of the consumer. Furthermore, the consumer RAC does not wait for the consumer’s acknowledgement before forwarding the next message to the consumer. Thus, communication between the consumer RAC and the consumer is asynchronous in both directions.

Based on this behavior of the consumer, the design of the consumer RAC is explained next. Every asynchronous message sent from a producer to the consumer passes through the Queue Coordinator of the Consumer RAC (Figure 4.2). When the Queue Coordinator receives a message from the producer, it queues the message into the Pending Queue (message 2a) and sends it to Connector Control (message 2). Connector Control then forwards the message to the consumer (message 3) and also forwards the message back to the Queue Coordinator (3a). As a result, the Queue Coordinator moves the message from the Pending Queue (3a.1a) to the Active Queue (3a.1b) which indicates that this message is currently being processed by the consumer. When the consumer finishes processing the producer’s message, it sends the ACK message (message 4) to the Connector Control of the consumer RAC. After Connector Control has received the ACK message from the consumer, it sends the Transaction Completed message (message 5) to the Queue Coordinator so that it removes the message from the Active Queue, which completes this transaction between the producer and the consumer.

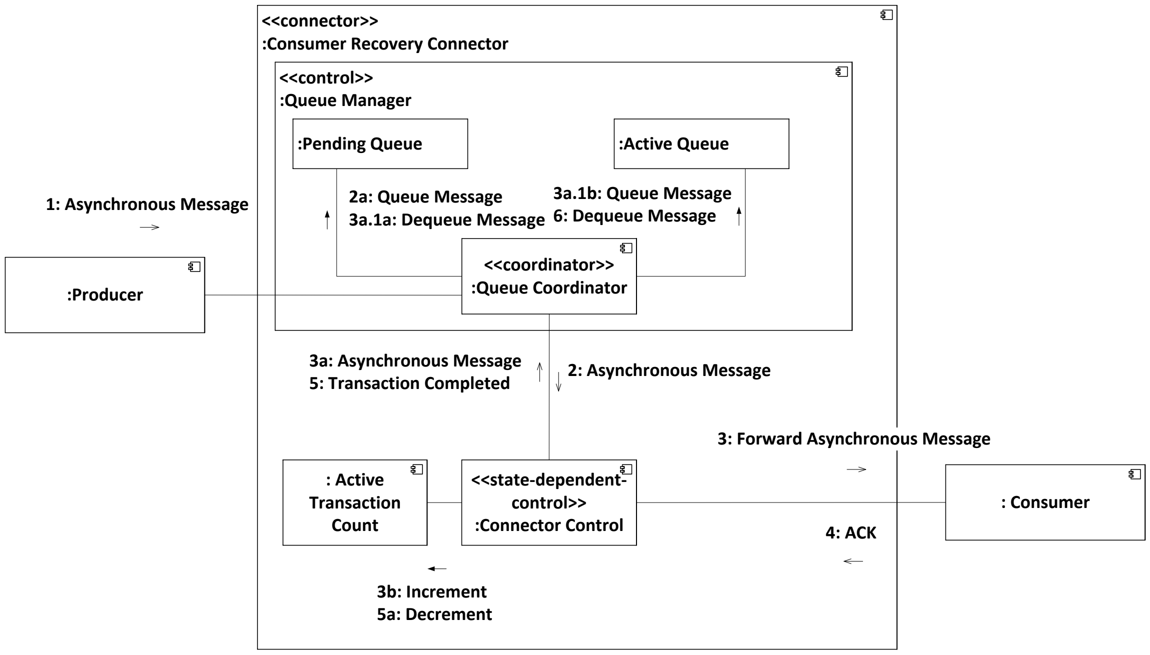


Figure 4.2 Design of the consumer RAC for the unidirectional asynchronous message communication

4.1.2 Connector Control State Machine and Normal Consumer Execution

The integrated adaptation and recovery state machine executed by connector control of the consumer RAC is shown in Figure 4.3. This state machine handles adaptation and recovery concerns for the consumer component. Initially, Connector Control is in the Waiting for Input Messages state. When Connector Control receives a producer's message, it (1) increments the transaction count, (2) forwards the message to the consumer, and then (3) transitions to the Processing state. When Connector Control receives a producer's message while in the Processing state, the actions are also to forward these messages to the consumer and to increment the transaction count. When Connector Control receives an ACK message from the consumer, it decrements the transaction count. Connector Control remains in the Processing state until it receives an acknowledgment from the consumer for every message it has forwarded so far and then transitions back to the Waiting for Input Messages state.

4.1.3 Dynamic Consumer Adaptation

In order to safely adapt the consumer at run-time, the consumer must be in the quiescent state in which it is not involved in any transactions and will not receive any new transactions from the producers. Thus, if Connector Control receives the Passivate command from Change Management (Kramer and Magee, 1990) while it is in the Waiting for Input Messages state (Figure 4.3), then the consumer is not engaged in any transactions with the producer. In this case, the consumer transitions immediately to the Quiescent state and notifies the Queue Coordinator of consumer quiescence so that it holds all messages it receives from the producer in the Pending Queue. On the other hand, if Connector Control receives the Passivate command while it is in the Processing state, then the consumer is engaged in one or more transactions with the producer. In this case, Connector Control must transition to the Passivating state in which the consumer is allowed to complete all transactions that it is currently engaged in with the producers. During the Passivating state, the consumer RAC does not forward any messages to the consumer. Thus, Connector Control notifies the Queue Coordinator so that it holds all message in the Pending Queue. Eventually, when Connector Control has received an ACK for every message that it has forwarded to the consumer, Connector Control transitions to the Quiescent state and notifies Change Management and Queue Coordinator that the consumer has become quiescent. When Connector Control is notified of consumer activation, then Connector Control notifies the Queue Coordinator to resume forwarding messages to Connector Control and then transitions to the Waiting for Input Messages.

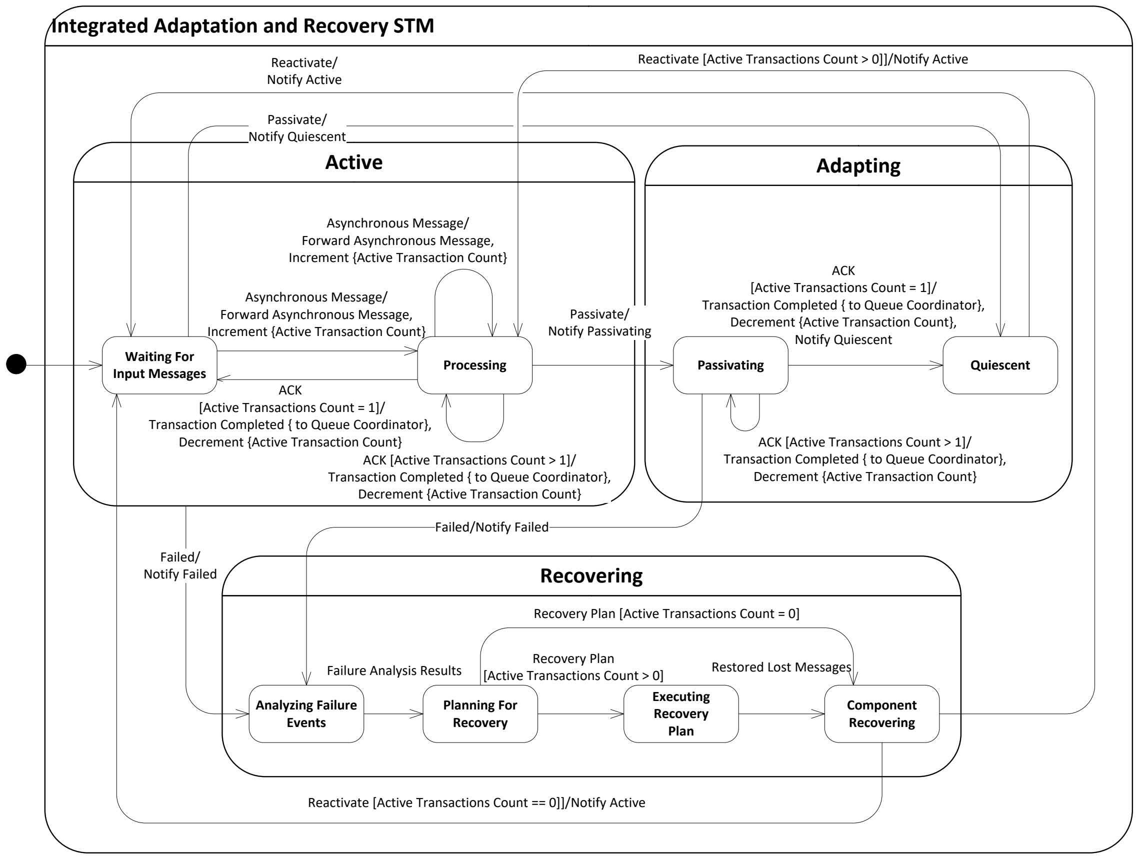

Integrated Adaptation and Recovery STM
Reactivate/ Notify Active
Reactivate [Active Transactions Count > 0]]/Notify Active
Passivate/ Notify Quiescent
Active
Adapting
Asynchronous Message/ Forward Asynchronous Message, Increment {Active Transaction Count}
Asynchronous Message/ Forward Asynchronous Message, Increment {Active Transaction Count}
Passivate/ Notify Passivating
ACK [Active Transactions Count = 1]/ Transaction Completed { to Queue Coordinator}, Decrement {Active Transaction Count}, Notify Quiescent
Waiting For Input Messages
Processing
Passivating
Quiescent
ACK [Active Transactions Count = 1]/ Transaction Completed { to Queue Coordinator}, Decrement {Active Transaction Count}
ACK [Active Transactions Count > 1]/ Transaction Completed { to Queue Coordinator}, Decrement {Active Transaction Count}
ACK [Active Transactions Count > 1]/ Transaction Completed { to Queue Coordinator}, Decrement {Active Transaction Count}
Failed/Notify Failed
Failed/ Notify Failed
Recovering
Recovery Plan [Active Transactions Count = 0]
Recovery Plan [Active Transactions Count > 0]
Failure Analysis Results
Restored Lost Messages
Analyzing Failure Events
Planning For Recovery
Executing Recovery Plan
Component Recovering
Reactivate [Active Transactions Count == 0]]/Notify Active

Figure 4.3 Integrated adaptation and recovery STM executed by connector control of the consumer recovery connector

4.1.4 Consumer Recovery

In the case of consumer failure, the recovery pattern is as follows:

1. When the consumer RAC is notified by the Recovery and Adaptation Manager (this manager will be discussed in chapter 5) of consumer failure, the consumer RAC transitions to the Analyzing Failure Events state (Figure 4.3) where it identifies any failed transactions and lost messages due to consumer failure. In this pattern, all messages queued in the Active Queue are considered lost messages since these messages are messages that have been forwarded to the consumer for which the consumer RAC has not received corresponding ACK messages from the consumer.
2. When all failed transactions have been identified, Connector Control transitions to the Planning for Recovery state where it determines the recovery plan for recovering failed transactions. In this pattern, Connector Control simply determines that every message queued in the Active Queue must be recovered and restored to the Pending Queue.
3. After the recovery plan is determined, Connector Control transitions to the Executing Recovery Plan state in which it executes the recovery plan by restoring all lost messages queued in the Active Queue to the head of Pending Queue.

When the consumer RAC is reactivated after it has been connected with the recovered consumer, the consumer RAC forwards the messages queued in the Pending Queue to the recovered consumer. This includes any held messages and lost messages that have been recovered according to steps 1-3.

4.1.5 Queue Coordinator State Machine

Figure 4.4 depicts the state machine executed by the Queue Coordinator. While in the Active state, the Queue Coordinator forwards all messages it receives from producers to Connector Control. When Queue Coordinator receives a notification that the Consumer is passivating, it transitions to the Passivating state, where it holds all producer messages in the Pending Queue. When the Consumer becomes quiescent, the Queue Coordinator receives the Notify Quiescent message from Connector Control and then transitions to the Quiescent state in which the action is to also hold all input messages in the Pending Queue until the consumer adaptation is completed.

In case of consumer failure, the Queue Coordinator eventually receives a consumer failure notification from Connector Control. As a result, the Queue Coordinator transitions to the Failed state in which it holds all producer messages in the Pending Queue. While in the Failed state, the Queue Coordinator may receive the Restore Request message from the Execution Activity of MAPE-K (as discussed in the previous section) to restore any lost messages. As a result, the Queue Coordinator restores these lost messages by moving these messages from Active Queue to Pending Queue. (Note that in order to ensure that all messages sent by the consumer before it failed are received by the consumer RAC, recovery of the failed consumer takes place only after pinging the node hosting the consumer component and then waiting for a certain time interval for receiving a heartbeat response. This issue of asserting failure of nodes using pinging is discussed in detail in chapter 5).

When the Queue Coordinator is notified that the consumer is activated again while in the Quiescent or Failed state, it forwards all messages held in the Pending Queue to Connector Control and then transitions to the Active state again.

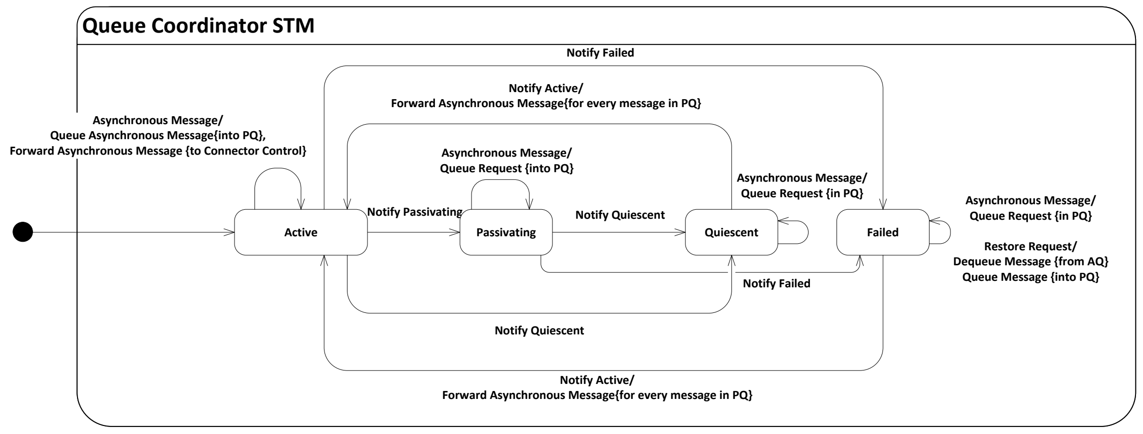


Figure 4.4 State machine executed by Queue Coordinator of the consumer RAC

4.1.6 Comparing the Designs of Consumer and Service RACs

Based on this communication pattern, the design of the consumer RAC described in this section is different from the service RAC for SOA patterns (chapter 3) as follows. First, since the consumer RAC does not receive any responses from the consumer, the consumer RAC (1) does not forward responses to the producer and (2) does not require the Service Response Manager that is used by the service RAC in SOA patterns to maintain responses from the service. Second, since the unidirectional asynchronous message communication pattern does not involve dialogs between the producer and consumer, the consumer RAC does not require the Service Recovery Queue, which is used by the service RAC to maintain previous requests for active dialogs between clients and the service.

4.2 Recovery of State-Dependent Consumers

In cases where the consumer is state-dependent, then the consumer must process every input message exactly once, even in the presence of failures. For instance, if the consumer is a component that handles the motor of a train and provides an interface to increase the speed of the train by a certain acceleration rate, then forwarding a message more than once to the consumer could cause a significant increase in the train speed beyond the intended speed, which could lead to catastrophic events.

It is assumed that the consumer maintains a log of the events it has executed (Elnozahy et al., 2002). Since the consumer is a state-dependent component, then the log is needed so that the consumer can recover its state after a run-time failure by replaying events from its log as well as to detect and discard duplicate messages (Elnozahy et al., 2002). The use of logs for state-dependent consumers instead of the two-phase commit (2PC) protocol (which is used in chapter 3 for a stateful service) is justified since (1) asynchronous patterns are widely used in real-time systems and (2) logging is more lightweight compared to the 2PC protocol and is thus more applicable to real-time systems. The consumer maintains this log as follows:

1. When the consumer receives a producer’s message that initiates a new transaction between the producer and the consumer, then the consumer logs the initiation of the transaction.

2. When the consumer is done with the producer’s message it adds a transaction completion record in its log and then sends the ACK message to the consumer RAC.

Recovery Pattern of State-Dependent Consumers

When the consumer recovers after a run-time failure, it uses its logs to guide its recovery as follows. First, during initialization, the recovered consumer replays messages from its log to recover its state (Elnozahy et al., 2002). Then, after reactivation, the consumer RAC forwards all lost messages to the recovered consumer, as explained previously in section 4.1.4. In case the consumer RAC sends duplicate messages to the recovered consumer, the recovered consumer detects and discards these duplicate messages by comparing the sequence number of input messages to the sequence number of messages maintained in the consumer log.

4.3 Recovery and Adaptation Patterns in Other Asynchronous Patterns

This section describes how the consumer RAC discussed in the previous section can also be used to handle adaptation and recovery of components in other asynchronous patterns.

4.3.1 Bidirectional Asynchronous Message Communication

In the bidirectional asynchronous message communication (Figure 4.5), the producer sends one or more asynchronous requests to the consumer via the consumer RAC. The consumer then processes each of these requests and sends a response to each request to the producer via the producer RAC. Furthermore, the producer can send a request to the consumer before it has received responses to the previous requests it has sent. This pattern is different from the asynchronous message communication with callback handle discussed in the previous chapter in that there can be more than one outstanding request at a time between the producer and consumer.

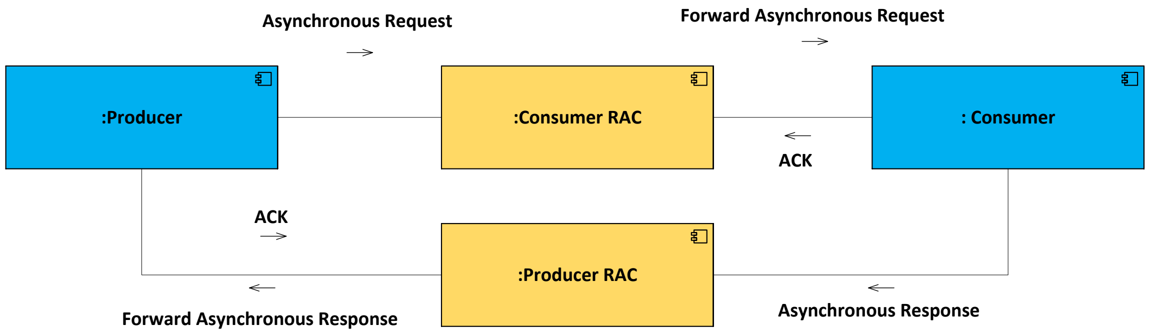


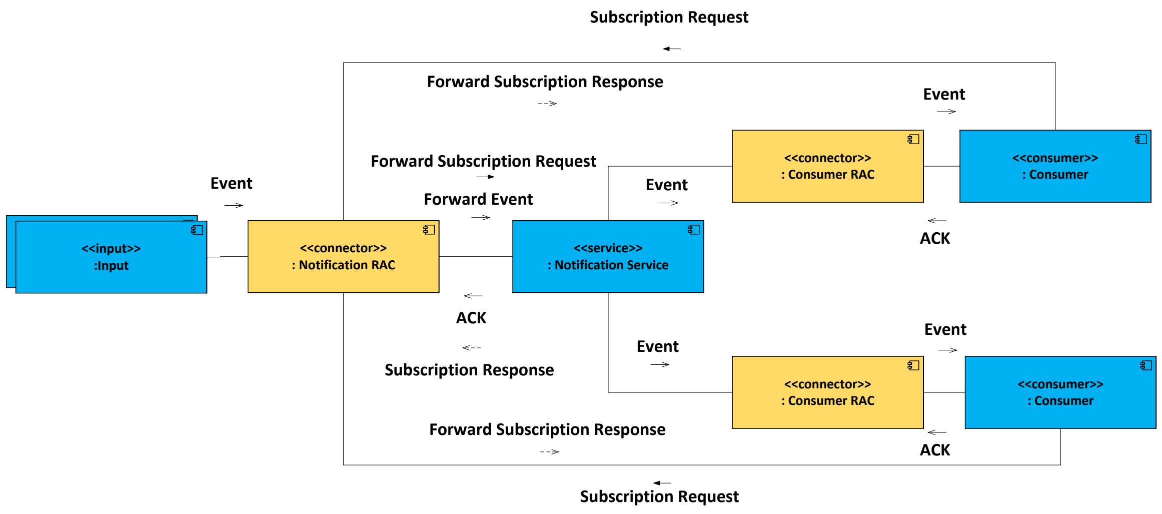
Figure 4.5 Bidirectional asynchronous message communication

This pattern is essentially a composition of two unidirectional asynchronous message communication patterns, since the producer sends asynchronous messages to the consumer in one direction without waiting for the consumer's responses and the consumer also sends asynchronous messages to the producer in another direction without waiting for the producer's messages. As a result, the adaptation and recovery patterns discussed in sections 4.1 and 4.2 apply equally to both the producer and the consumer engaged in bidirectional asynchronous message communication.

4.3.2 Subscription/Notification Pattern

The subscription/notification pattern (Figure 4.6) is a selective form of group communication in which consumers subscribe to a notification service to receive events of a certain type. To facilitate this type of communication, a notification service is provided to handle tracking of consumers and the type of events that each consumer is interested in. Consumers can subscribe (or unsubscribe) to the notification service and specify the type of events they need to receive. When the notification service receives an event, it multicasts this event to the consumers that have subscribed to receive this type of event. This pattern consists of two type of transactions:

- subscription (or unsubscription) of consumers with the notification service.
- multicast of events by the notification service to consumers.



Subscription Request
Forward Subscription Response
Event
Forward Subscription Request
Event
Forward Event
Event
<<connector>> : Consumer RAC
<<consumer>> : Consumer
ACK
<<input>> :Input
<<connector>> : Notification RAC
<<service>> : Notification Service
ACK
Subscription Response
Event
Event
<<connector>> : Consumer RAC
<<consumer>> : Consumer
Forward Subscription Response
ACK
Subscription Request

Figure 4.6 Subscription/notification pattern

Transactions involving subscription (or unsubscription) of consumers use the synchronous message communication with reply pattern between each consumer and the notification service. Therefore, the recovery and adaptation pattern for this type of transaction is exactly the same as the one discussed previously in chapter 3.

Recovery of Consumers

Since the communication between the notification service and *each* consumer is unidirectional, then each consumer is associated with a consumer RAC that handles receiving the multicast message from the notification service and then forwarding this message to the consumer. As a result, the adaptation and recovery patterns for consumers in this pattern are exactly the same as shown previously in section 4.1 for unidirectional asynchronous message communication.

Recovery of Notification Service

In this pattern, events to the notification service are forwarded by the service's RAC as follows. During normal execution, the Notification RAC forwards the events it receives to the notification service. When the notification service receives an event from its connector, it determines which consumers have registered to receive this type of event and then notifies these consumers of the event. When the notification service sends the event to all such consumers, it sends the ACK message (Figure 4.6) to its connector so that the connector removes the event from its queues. Note that this pattern ensures that the notification is not lost by the notification service, but does not guarantee that the notification reaches the consumer or is processed by the consumer, because this is a lightweight protocol and hence does not use transactions.

Based on this behavior, the Notification RAC recovers all lost unidirectional messages to the notification service using the recovery pattern for the consumer in the unidirectional asynchronous message communication discussed in section 4.1.

4.3.3 Master/Slave Pattern

In the master/slave pattern (Figure 4.7), the master component is responsible for serving requests it receives from clients by dividing up the task to be performed among multiple slaves. The master sends a command to each slave via the slave's RAC specifying the part to be executed by the slave. The slaves then process the master's commands in parallel. When a slave finishes processing the master's command, it sends a response to the master via the master RAC. Finally, the master integrates the slave responses.

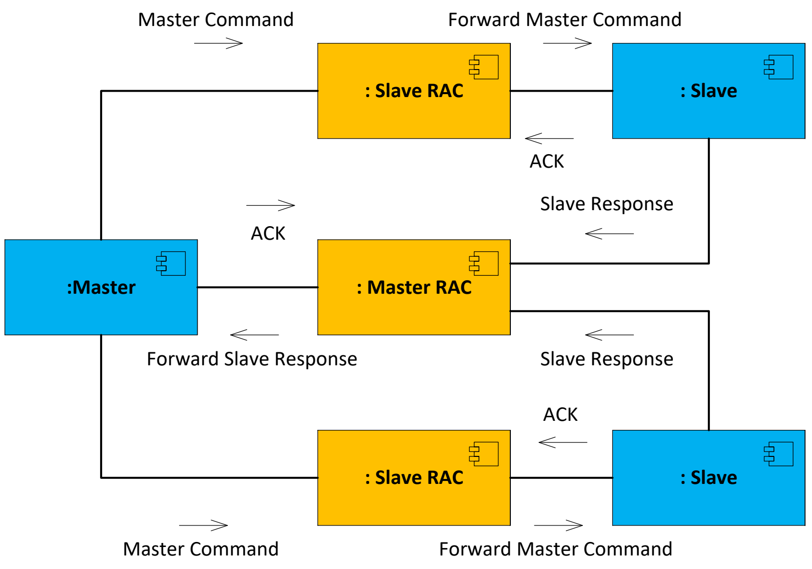


Figure 4.7 Master/slave pattern

<u>Recovery and Adaptation of Slaves</u>

In this pattern, the behavior of slaves is similar to the behavior of the consumer in the bidirectional asynchronous message communication (c.f. Figure 4.5). As a result, the recovery and adaptation pattern for slaves is exactly the same as the recovery and adaptation pattern for consumers in the bidirectional asynchronous message communication discussed in section 4.2.

Recovery and Adaptation of Master

In this pattern, we consider that the master interacts with its slaves as a compound transaction (Gomaa, 2011) that can be broken down into multiple atomic transactions between the master and each slave. The master first initiates a compound transaction and then sends a command to each slave within this compound transaction (Figure 4.8). When the master RAC receives a slave response, it forwards the response to the master. The master then (1) integrates all slave responses and sends an ACK message to the master RAC so that it removes the slave responses from its queue.

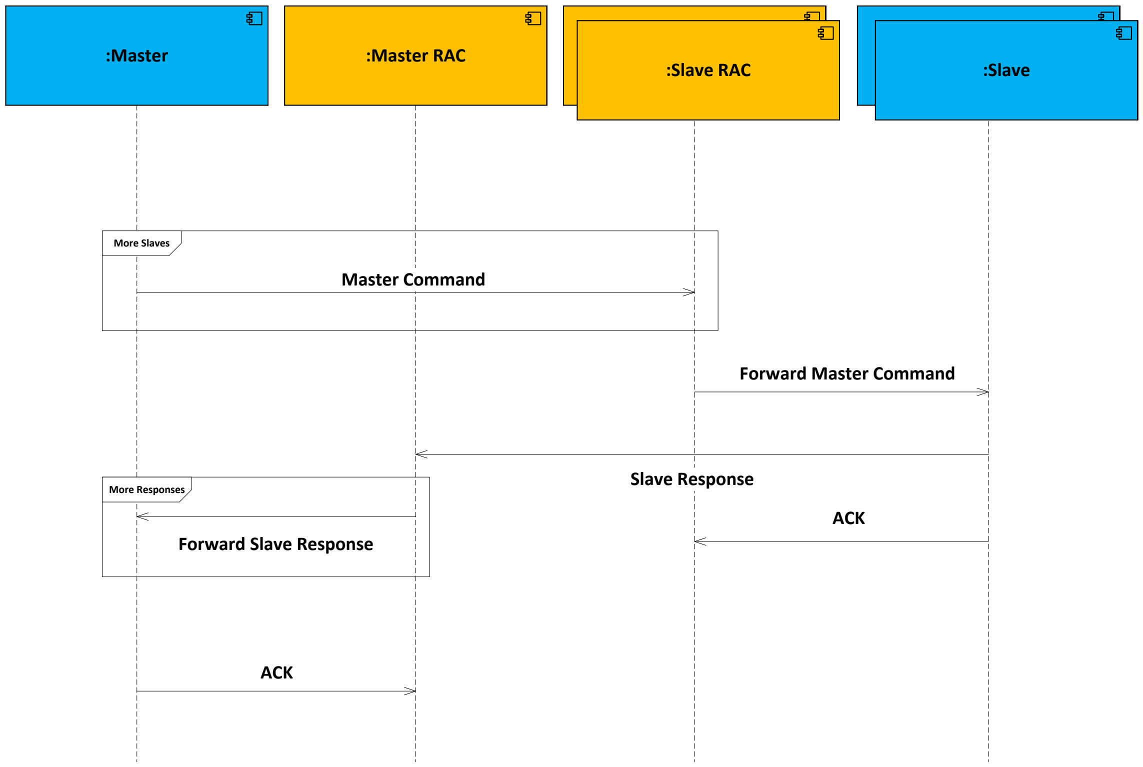


Figure 4.8 Message sequencing between master and slaves

To facilitate the recovery of the master after a run-time failure, it is assumed that the master maintains a log to keep track of the compound transactions it initiates. The log is used to ensure that if the master fails after initiating a compound transaction but before this transaction is completed, then the recovered master can restart this compound transaction using its log. However, the use of the log alone does not ensure recovery of lost messages that were forwarded to the master either before or after it failed. Therefore, the master RAC is used to recover and resend any such lost messages to the recovered master, as explained next.

For each compound transaction, the master logs two records as follows:

1. Before initiating any transaction with the slaves, the master first logs the initiation of the compound transaction.
2. Before sending the ACK message to the master RAC, the master logs the completion of the compound transaction. The master also includes in this log record the result of integrating all of the slave responses it has received. Thus, after this record has been added to the log, the master RAC can safely remove all slave responses from its queues.

When the master recovers from a run-time failure, it can be in two states, as follows:

1. The master's log indicates the initiation of an incomplete compound transaction. In this situation, the master must have failed before integrating the slave responses for the compound transaction it has initiated. Thus, the recovered master restarts this compound transaction. Due to this, a slave RAC may receive a duplicate command from the recovered master. Thus, slave RACs detect and discard these duplicate commands using message sequence numbers. Finally, the master RAC restores any slave responses it has forwarded to the master by moving these responses from the Active Queue to the head of the Pending Queue, as discussed previously in section 4.1. Note that the master RAC could not have removed any slave responses for this transaction from its queues since the master failed before sending the ACK message to the master RAC.
2. The master's log indicates that the last compound transaction has been completed. In this situation, there are two cases to consider: the master could have failed either before or after sending the ACK message to the master RAC. If the master

failed after it has sent the ACK message to the master RAC, then no recovery actions are required from the recovered master or the master RAC. Otherwise if the master failed before sending the ACK message to the master RAC, then the master RAC will recover and resend the slave responses of this transaction to the recovered master. Since the master maintains a log of the transactions it initiates, then the master can use this log to (1) detect and discard duplicate slave responses from the master RAC by comparing the transaction identifier of these responses to the transaction identifier of previously completed compound transactions and (2) send the ACK message to the master RAC so that it removes the slave responses from its queue.

4.3.4 Centralized Control Pattern

The centralized control pattern is widely used in real-time, embedded systems in which there is a centralized control component that handles control of other components in a software system (or a subsystem). This centralized control component (Figure 4.9) is responsible for receiving input events from multiple input components and sending output commands to output components. The centralized control component is state-dependent and encapsulates a state machine that specifies the output commands (known as *actions*) that must be sent to output devices based on (1) the current state of the control component and (2) the received input event.

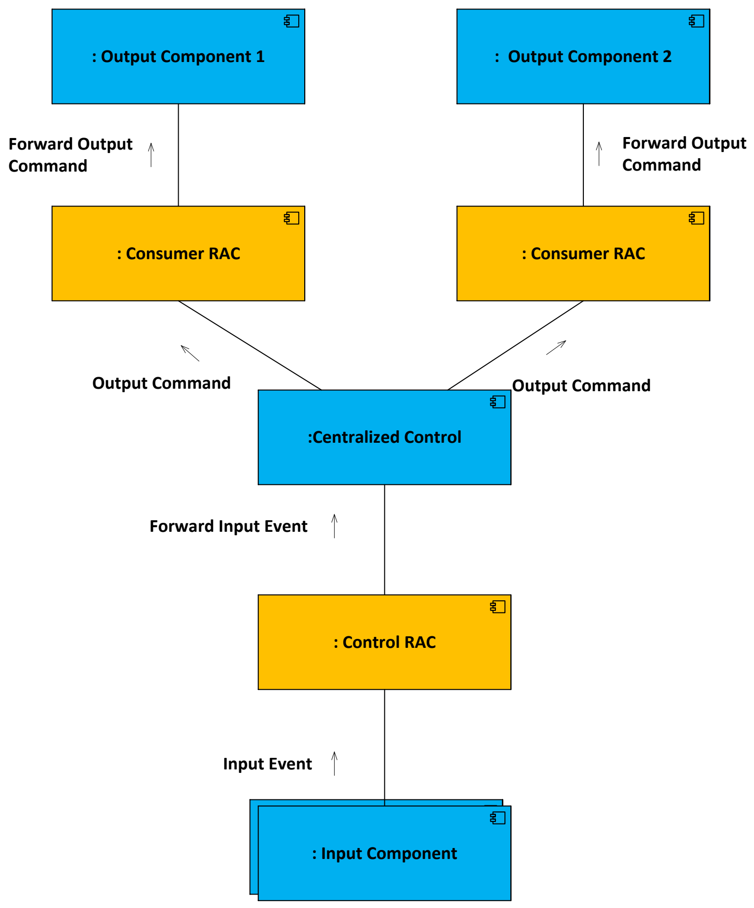


Figure 4.9 Centralized control pattern

In this pattern, communication between the centralized control component and each output component uses the unidirectional asynchronous message communication pattern. Thus, each output component is associated with a consumer RAC that handles the recovery of its component as discussed previously in section 4.1.

Since the communication between each input component and the centralized control component is also based on the unidirectional asynchronous message communication pattern, then the centralized control component is associated with a consumer RAC (control RAC in Figure 4.9) that executes the state machine shown previously in Figure 4.3.

Since the centralized control component is state-dependent, it is assumed that the state-dependent control component maintains an execution trace log that can be used by the component to guide its recovery after a run-time failure. Since the control component is a state-dependent component, then the log is needed to ensure that the control component can reconstruct its state after recovering from a run-time failure by replaying logged events (Elnozahy et al., 2002). The log is updated by the centralized control component during normal execution each time the component receives an input event from its connector by logging:

1. the input event.
2. the state transition (source and destination states of the transition) caused by this input event.
3. the actions performed by the state-dependent control component as a result of the state transition.
4. A Completed record indicating that this state transition and all actions associated with this state transition have been performed.

After the centralized control component logs completion of the transaction, it sends ACK message to its RAC so that it removes the input event from its queues.

When the centralized control component recovers from a run-time failure, it reconstructs the state of the state machine by replaying input events in its log. During this process, the recovered centralized control component may send duplicate commands to output components. These duplicate commands are detected and discarded by the RACs of these output components using message sequence number.

Note that the centralized control component could have failed before logging some input events that have been forwarded by the central control RAC. In this case, the control RAC ensures that these events (which have been lost by the control component due to failure) are resent to the recovered control component along with any events held while the component is in the failed state, as discussed in section 4.1. Since the centralized control component maintains a log of the events it has received, the component can use its log to detect and discard any duplicate input events it may receive from its connector.

4.3.5 Distributed Control Pattern

The distributed control pattern (Figure 4.10) is used in more complex distributed systems in which there are multiple state-dependent control components (instead of one centralized control component as done in the centralized control pattern) such that each control component manages a different part of the software system than other control components. Furthermore, one control component can interact with other control components. In this pattern, there is no single control component that has an overall control of the system.

In this pattern, a control component M interacts with its predecessor control component M-1 and successor control component M+1 as follows (Figure 4.10):

1. Control component M initiates a transaction with its predecessor M-1 by sending it an asynchronous request and then receiving the corresponding asynchronous response later.
2. Control component M receives a transaction from its successor M+1 by receiving an asynchronous request from its successor and then sending the corresponding asynchronous response later.

As there are many variations to this pattern, we consider in this research that a control component M can initiate a transaction with its predecessor M-1 only after control component M has received a response for the last request it sent to control component M-1. Thus, a control component in this pattern is involved in different communication patterns and roles as follows:

1. A control component M can act as a consumer in the unidirectional asynchronous message communication by receiving input events from input components.
2. A control component M can act as a producer in the unidirectional asynchronous message communication by sending output events to output components.
3. A control component M can act as a consumer in the asynchronous message communication with callback pattern (see section 3.3.1 in chapter 3) by receiving an asynchronous request from another control component and then sending an asynchronous response to this component.
4. A control component M can act as a producer in the asynchronous message communication with callback pattern (see section 3.3.1 in chapter 3) by sending

an asynchronous request to another control component and then receiving the corresponding asynchronous response from this component.

In this research, we consider that receiving an input message from an input component causes a distributed control component to initiate a transaction with another control component as follows:

1. When the RAC of a control component M receives a message from an input component, this RAC forwards the input message to control component M.
2. Control component M then sends an asynchronous request to another control component M-1 using the asynchronous message communication with callback pattern.
3. Eventually, control component M-1 sends an asynchronous response to control component M.
4. When control M is done with the response, control M sends an ACK message to its RAC, which serves as an indication to the control RAC that the control component is done with the input message sent in step 1.

Based on this behavior of control components, a control component can be adapted after it has sent (1) an ACK message for each unidirectional asynchronous message it received from its RAC and (2) an asynchronous response for each asynchronous request it received from its RAC. In case of the failure of a control component, then the RAC of this control component resends all (1) unacknowledged unidirectional asynchronous messages and (2) any asynchronous requests for which there is no corresponding response received at the RAC to the recovered control component.

Therefore, the RAC of each control component in this pattern must be able to handle integration of unidirectional asynchronous message communication and asynchronous message communication with callback. This problem of patterns integration at the RAC will be discussed in detail in chapter 8.

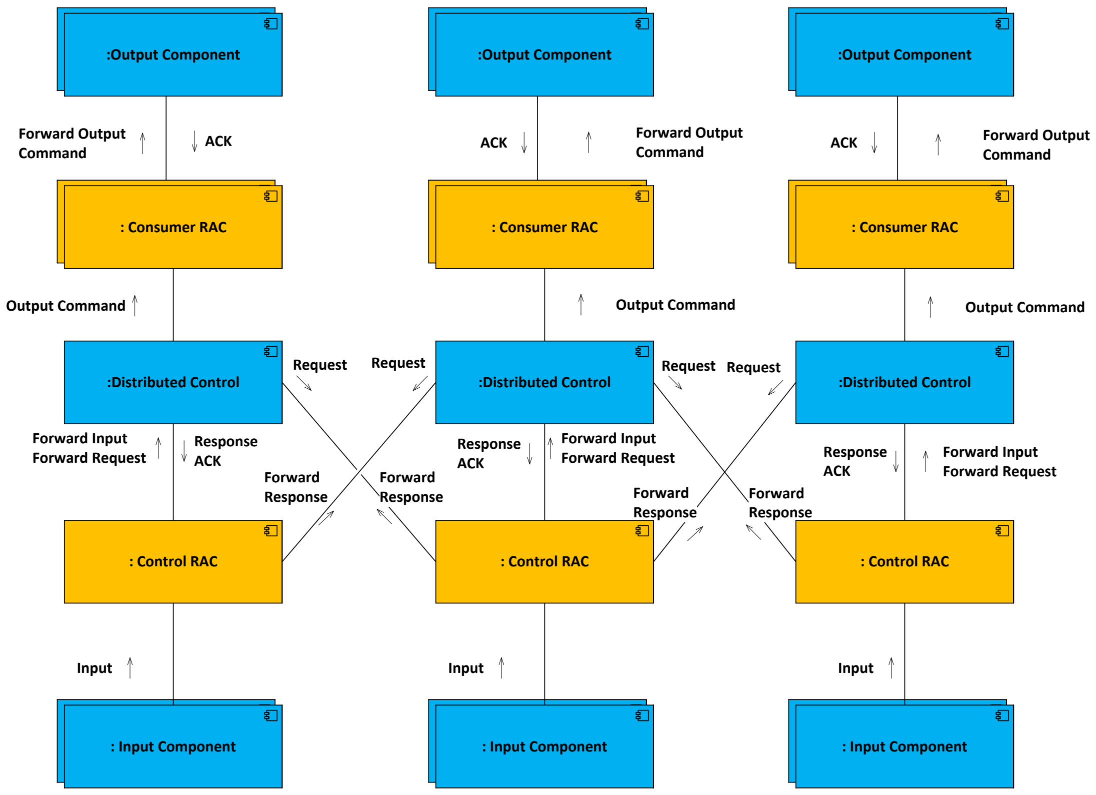


Figure 4.10 Distributed control pattern

4.3.6 Hierarchical Control Pattern

In the hierarchical control pattern (Figure 4.11), there are multiple distributed control components, each controlling a different part of the software system. However, unlike the distributed control pattern, the distributed control components in this pattern are controlled by a higher-level control component (Hierarchical Control in Figure 4.11) which decides the next job for each low-level control component. As there are many variations of this pattern, we assume that the hierarchical control component receives new job messages from one or more job generator components (e.g. producers or external systems). For each new job, the hierarchical control component assigns the job to one of the low-level control components by sending it a command. When a low-level control component finishes the current job assigned to it, it sends a response to the hierarchical control component. As a result, the hierarchical control component can send this low-level control component the next command. It is assumed that the hierarchical control component does not send a command to a low-level control component unless the hierarchical control component has received a response for the previous request it has sent to the low-level control component.

In this pattern, since the hierarchical control component receives asynchronous messages from generator components using the unidirectional asynchronous message communication, then the adaptation and recovery patterns of the hierarchical control component are the same as the ones shown previously in section 4.2.

On the other hand, every low-level distributed control component participates in two patterns:

- The unidirectional asynchronous message communication since a low-level control component may receive asynchronous messages from input components and sends asynchronous messages to output components.

- The asynchronous message communication with callback pattern since a distributed control component receives asynchronous commands from the hierarchical control and sends the corresponding responses of these commands to the hierarchical control component.

Therefore, similar to the distributed control components in the distributed control pattern (see the previous section), the RAC of each low-level control component in the hierarchical control pattern must be able to handle integration of unidirectional asynchronous message communication and asynchronous message communication with callback. This problem of patterns integration at the RAC will be discussed in detail in chapter 8.

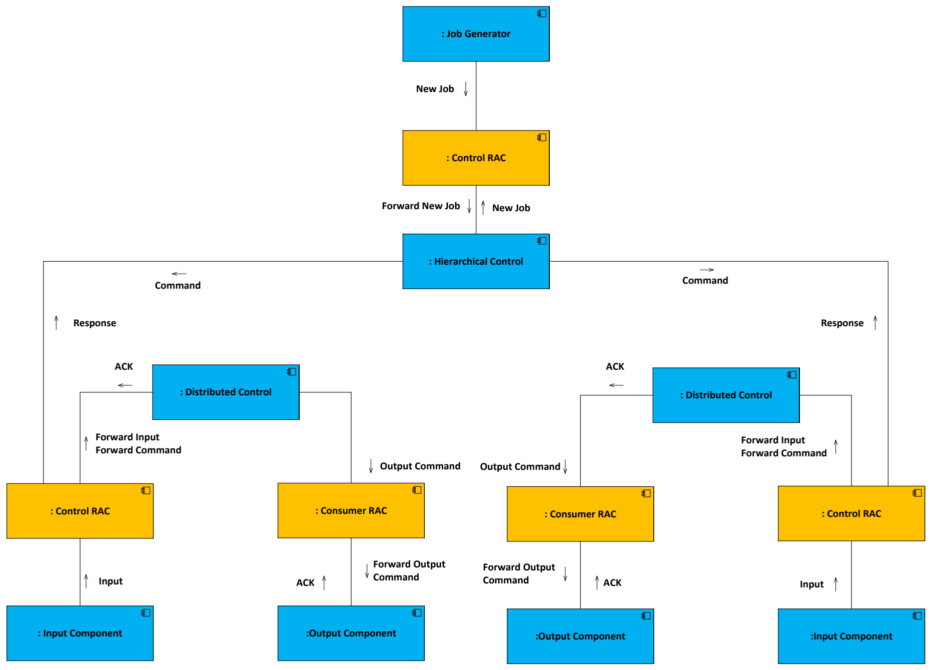


Figure 4.11 Hierarchical control pattern

5 DESIGN OF DECENTRALIZED ARCHITECTURE FOR SELF-HEALING AND SELF-CONFIGURATION

The focus of the previous chapters was recovering failed *transactions* using recovery and adaptation connectors (RACs) in various architectural patterns. An equally important problem is recovering *components* automatically after a run-time node failure, which is the focus of this chapter.

When a node fails, the system must recover to a consistent configuration in which every failed component is relocated and instantiated on a healthy (i.e. non-failed) node and that the connections between a recovered component and its neighbor components are re-established.

This chapter is organized as follows. Section 5.1 discusses DARE, a decentralized, architecture-based framework for self-healing and self-configuration, which is based on a decentralized MAPE-K loop model. Sections 5.2 and 5.3 describe the design of the various components in the DARE architecture. Section 5.4 describes the mapping of recovery functionality to MAPE-K activities. Section 5.5 describes how DARE can be used to recover failed components.

5.1 DARE Overview

This section provides an overview of the DARE (Distributed Adaptation and REcovery) framework, which is a decentralized, integrated adaptation and recovery framework for providing both self-healing and self-configuration properties to complex and highly dynamic CBSAs. In this dissertation, every node in the system hosts an identical instance of the DARE middleware whose architecture is shown in Figure 5.1. This architecture consists of three layers as follows.

The Configuration Maintenance Layer (CML) is responsible for keeping track of the current configuration map of the software system, which includes the mapping of components to nodes, and providing services to higher layers for retrieving and modifying this map.

The Architecture Discovery Layer (ADL) is responsible for automatically discovering the current architecture of the software system. It relies on gossiping and message tracing techniques for discovering and disseminating the current software architecture between nodes in a decentralized fashion (Porter et al., 2016). Furthermore, the ADL is responsible for notifying the CML, when it suspects a node failure due to absence of gossip messages from that node, and providing the discovered architecture to the top layer when dynamic adaptation and failure recovery are required. The design and implementation of DeSARM is beyond the scope of this dissertation.

The Application Recovery Layer (ARL) is responsible for adapting and recovering components after a run-time component failure. The Recovery and Adaptation Manager in this layer determines plans for dynamically adapting the architecture and recovering failed nodes. Additionally, this manager executes a reconfiguration template consisting of reconfiguration commands that handle instantiating components on healthy nodes and establishing the connections between application components. This layer also includes recovery and adaptation connectors (not shown in Figure 5.1) that handle recovery of failed transactions and steer application components to a quiescent state in order to carry out dynamic adaptation as shown previously in chapters 3 and 4.

The next two sections describe the design of the CML and ARL. Interested readers can refer to (Porter et al., 2016) for more details on the design of the ADL.

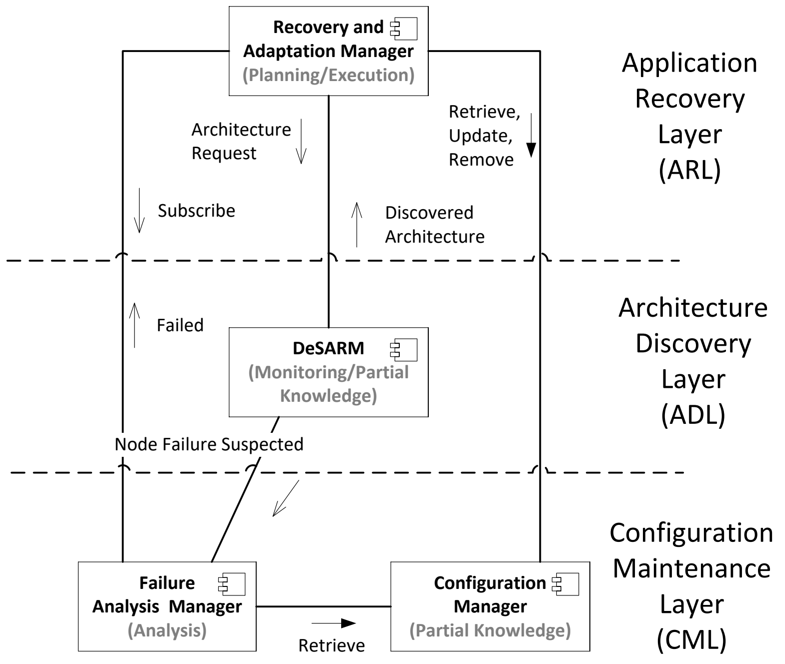


Figure 5.1 The DARE architecture

5.2 Configuration Maintenance Layer

The Configuration Maintenance Layer (CML) consists of the Configuration Manager and the Failure Analysis Manager, which are described next.

5.2.1 Configuration Manager

The Configuration Manager (CM) is responsible for maintaining the current configuration map of the software system, which includes:

- The IP and subnet address of every node in the current configuration of the software system, and
- The set of identifiers of components and RACs hosted by every node in the configuration.

In order to tolerate failures and enable distribution of the configuration map, the configuration map is stored in a distributed hash table (DHT) that supports replication of its entries (Stoica et al., 2003). The DHT contains entries (see Table 5.1) that map (1) the IP address of a node to the set of identifiers of components and RACs hosted by the node with this IP address, (2) the identifier of a component or RAC to the IP address of the node that is currently hosting this component or RAC, and (3) a subnet address to the IP addresses of the nodes that are currently in this subnet.

Table 5.1 Key and value pairs of the distributed hash table used by configuration manager to store the configuration map

Key (hash of)	Value
Node IP Address	Set of identifiers of the components and connectors hosted by the node with this IP address.
Component/Connector ID	The address of the node hosting the component or connector with this identifier.
Subnet Address	Set of the IP addresses of the nodes in this subnet

	address.

5.2.2 Failure Analysis Manager

At any one moment, the recovery of a failed component must be handled by exactly one Recovery and Adaptation Manager. Otherwise, if multiple managers attempted to recover the same failed component, the system's configuration may become inconsistent with, for example, duplicate components and broken connections between components. To ensure consistent recovery, our approach involves electing the node with the lowest IP address to become the *recovery coordination node* in charge of coordinating recovery of other failed nodes. To ensure scalability of the approach in case of large systems that span multiple subnets, it is possible to have a recovery node for each subnet such that this recovery node handles recovery of other nodes in the same subnet. The Failure Analysis Manager (FAM) module in the recovery node is the *only* FAM that proceeds with the recovery process by analyzing the failure, as described next.

The state machine executed by the FAM is shown in Figure 5.2. Initially, the FAM is in the Idle state. When the FAM receives a notification message from DeSARM that a node failure is suspected, the FAM first retrieves from the Configuration Manager (CM) the IP addresses of all nodes that are in the same subnet as the node hosting this FAM and then transitions to the Determining Recovery Node state. The FAM then checks whether the suspected node belongs to this subnet. If the suspected failed node is not in the same subnet as the node hosting this FAM, then the FAM transitions back to the Idle state. Otherwise, if both nodes belong to the same subnet, then the FAM determines the lowest IP address in this subnet using the set of IP addresses it obtained from the CM and then checks whether it is hosted by the node with this IP address. At this point, the recovery node has been determined and only the FAM hosted by the node with the lowest IP address proceeds with the recovery process by pinging the suspected failed node and then transitioning to the Waiting for Heartbeat Message state (Figure 5.2). If a heartbeat message is received from the suspected node, then this node is running normally. In this case, the FAM transitions to the Idle state. Otherwise, if no heartbeat message is received within a certain time interval, then the FAM at the recovery node (1) notifies the peer Recovery and Adaptation Manager (which is located at the same recovery node as this FAM) of a node failure so that it handles recovery of components deployed to that node and then (2) transitions to Idle state.

This approach ensures that a recovery node is always elected, even if the node that failed has the lowest IP address in the subnet, for two reasons. First, DeSARM always ensures that the FAM on every healthy node will receive a notification that a node failure is suspected. Second, when a FAM determines the lowest IP address of a subnet, it always removes the IP address of the suspected failed node from the set of addresses it obtained from the CM (as shown in Figure 5.2), so that if the failed node has the lowest IP address in the subnet, then the node with the next lowest IP address can be selected to become the recovery node.

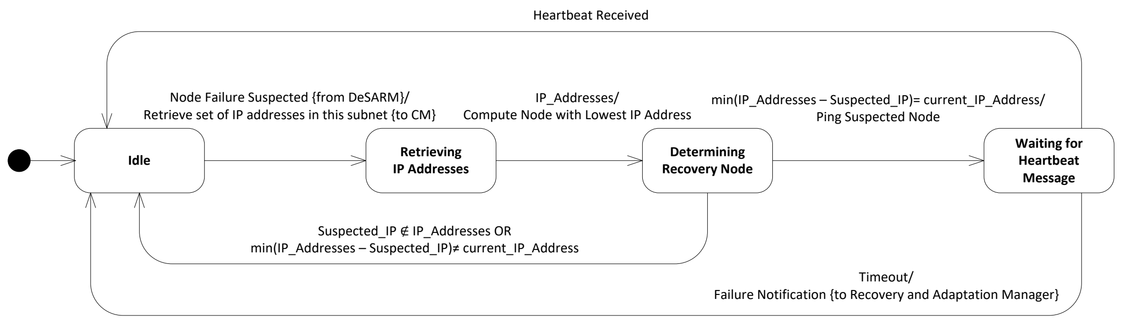

Heartbeat Received
Node Failure Suspected {from DeSARM}/
Retrieve set of IP addresses in this subnet {to CM}
IP_Addresses/
Compute Node with Lowest IP Address
min(IP_Addresses – Suspected_IP)= current_IP_Address/
Ping Suspected Node
Idle
Retrieving IP Addresses
Determining Recovery Node
Waiting for Heartbeat Message
Suspected_IP ∉ IP_Addresses OR
min(IP_Addresses – Suspected_IP)≠ current_IP_Address
Timeout/
Failure Notification {to Recovery and Adaptation Manager}

Figure 5.2 FAM STM

5.3 Application Recovery Layer

The Application Recovery Layer (ARL) is responsible for overseeing dynamic adaptation of the CBSA. Furthermore, when one or more nodes fail, the ARL ensures that the software system recovers to a consistent configuration in which every failed component is relocated and instantiated on a healthy (i.e., non-failed) node and that the connections between a recovered component and its neighbor components are re-established. The ARL consists of Recovery and Adaptation Connectors (see chapters 3 and 4) and Recovery and Adaptation Managers. This section describes the design of the Recovery and Adaptation Manager.

The Recovery and Adaptation Manager (RAM) is responsible for planning and executing dynamic adaptation and failure recovery. The RAM executes the state machine shown in Figure 5.3, which defines behavior during (1) failure recovery and (2) dynamic adaptation, as explained next.

5.3.1 Behavior of Recovery and Adaptation Manager During Failure Recovery

When the RAM at the recovery node receives a failure notification from the FAM while in the Idle state (Figure 5.3), it requests the current architecture from DeSARM located at the same node and then transitions to the Requesting Architecture state. When the RAM receives the architecture, it transitions to the Determining Recovery Plan state in which it determines the recovery plan as explained next.

First, the RAM retrieves from the CM the set of component identifiers deployed to the failed node. The RAM then uses the information it obtained from DeSARM and the CM to determine a recovery plan for *each* component hosted by the failed node as follows:

- The RAM first determines the RAC of the failed component by looking up the architecture it obtained from DeSARM for any RAC that sends input messages, e.g., synchronous requests or asynchronous messages, to the failed component. The RAM then notifies this *input* RAC of component failure. As a result, each affected input RAC ceases forwarding messages to its component and begins recovering failed transactions.
- The RAM determines the RACs of other application components that receive input messages from the failed component so that the recovered component can also be connected with these *recipient* RACs. The RAM determines these recipient RACs by looking up the architecture it obtained from DeSARM for any component that receives either synchronous requests or asynchronous messages from the failed component.
- The RAM determines the node to host the recovered component. In our current design, the RAM selects any random node for this purpose. Alternatively, a self-

optimization approach (Menasce et al., 2011) could be used to select the optimal node for hosting this component.

After the recovery plan is determined, the RAM transitions to the Performing Recovery state (Figure 5.3) in which it proceeds with recovery by executing a reconfiguration template to recover each component according to the recovery plan. The recovery template consists of reconfiguration commands (Kramer and Magee, 1990) for restoring the system to a consistent configuration through the following RAM actions:

- Instantiate another instance of each failed component on a different node according to the recovery plan.
- Connect the RACs that communicate with the recovered component by sending them the Connect command along with the address of the recovered component.
- Update the configuration map so that it reflects the new location of the recovered component. The RAM also updates the configuration map by removing the failed node from the configuration and by adding the new node to the configuration.
- Activate the recovered component by sending it the Activate command.
- Send a Reactivate command to the input RAC of the recovered component so that it resumes normal communication with the recovered component. As a result, the input RAC resumes forwarding messages to the recovered component, including any messages that have been lost due to component failure.

Once recovery is complete, the RAM transitions back to the Idle state. The recovery process is distributed among the RAM and the RAC (Figure 5.4). While in the Determining Recovery Plan state, the RAM at the recovery coordination node (node R) notifies the RAC of component failure and continues execution. This RAC is hosted at a different node (node X). As a result, this RAC starts recovering failed transactions concurrently while the RAM recovers the failed component. While in the Performing Recovery state, the RAM at node R then coordinates component recovery by sending the Create command to the RAM at node Y that will host the recovered component. Eventually, the RAM at node R receives an acknowledgement that the application component is created. The RAM then requests the RAC to connect with the recovered component at Node Y. Finally, the RAM activates the recovered component and reactivates the RAC so that it forwards any lost messages to the recovered component.

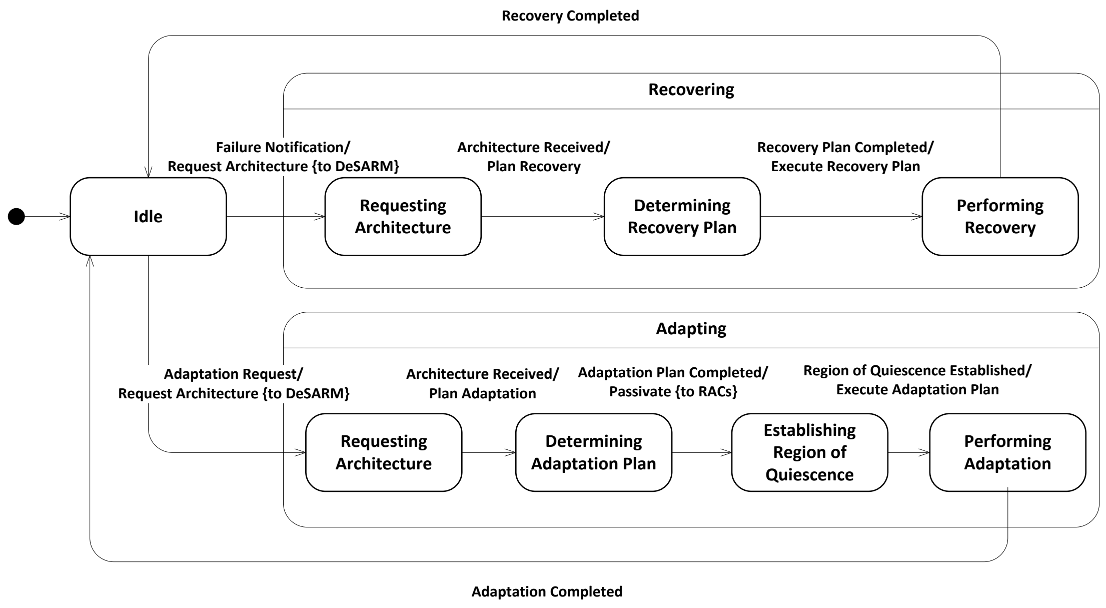

Recovery Completed
Recovering
Failure Notification/ Request Architecture {to DeSARM}
Architecture Received/ Plan Recovery
Recovery Plan Completed/ Execute Recovery Plan
Idle
Requesting Architecture
Determining Recovery Plan
Performing Recovery
Adapting
Adaptation Request/ Request Architecture {to DeSARM}
Architecture Received/ Plan Adaptation
Adaptation Plan Completed/ Passivate {to RACs}
Region of Quiescence Established/ Execute Adaptation Plan
Requesting Architecture
Determining Adaptation Plan
Establishing Region of Quiescence
Performing Adaptation
Adaptation Completed

Figure 5.3 RAM state machine

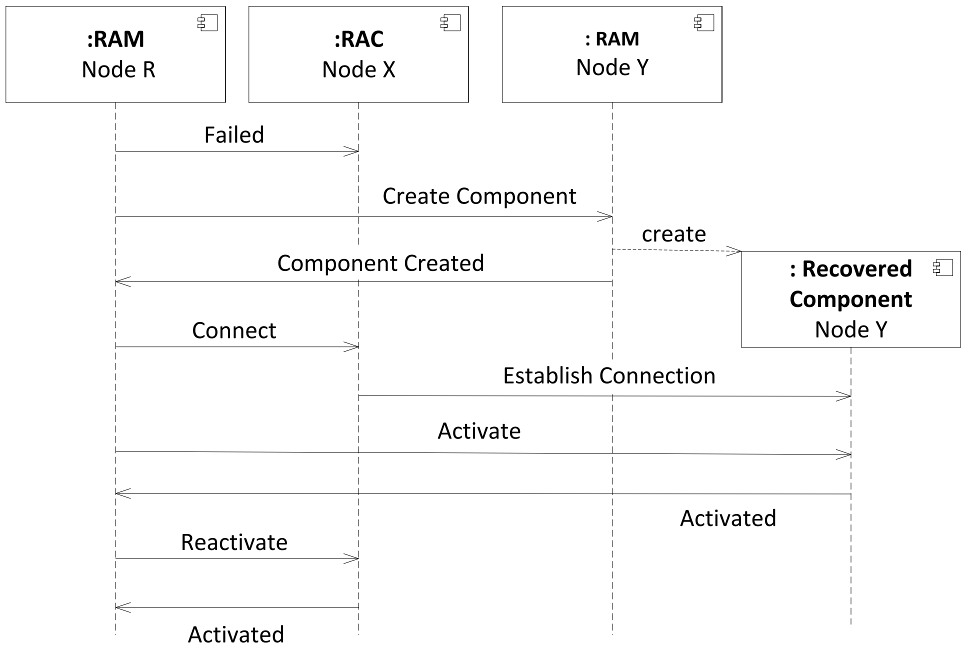


Figure 5.4 Fragment of the distributed recovery process

5.3.2 Behavior of Recovery and Adaptation Manager During Dynamic Adaptation

While in the Idle state, the RAM can receive an external adaptation request (Menasce et al., 2011) to add a new application component to the architecture, remove an application component from the architecture or, replace one component with another.

To carry out dynamic adaptation, the RAM needs the current software architecture, which it requests from DeSARM before transitioning to the Requesting Architecture state. When the RAM receives the architecture, it transitions to the Determining Adaptation Plan state in which the RAM determines the (1) input RAC of the application component affected by the adaptation, (2) the recipient RACs that receive input messages from the application component affected by the adaptation, and (3) the node that will host the new component (in case of adding a new application component to the architecture), similarly to how it is done during recovery.

Once planning is complete, the RAM establishes a region of quiescence (Kramer and Magee, 1990) by sending the Passivate command to the input RAC of the application component affected by dynamic adaptation so that this RAC drives the application component to a quiescent state, in which this component is not engaged in any transactions and will not receive any new transactions from other application components. When the application component is quiescent, the component's RAC sends the quiescence notification to the RAM. As a result, the RAM starts adapting the architecture and transitions to the Performing Adaptation state.

While in the Performing Adaptation state, the RAM executes an adaptation template that consists of reconfiguration commands to (1) create any new application component that needs to be added to the architecture, (2) disconnect and remove application components that need to be removed from the architecture, (3) connect new application components with the RACs, (4) update the configuration map to reflect the new configuration of the software architecture, (5) notify DeSARM of architecture adaptation so that it can start the process of discovering the new architecture, and (6) activate application components and input RACs. After dynamic adaptation is complete, the RAM transitions back to the Idle state.

5.4 Mapping Recovery Functionality to MAPE Activities

The recovery process described in the previous section is based to the MAPE-K loop model (Kephart and Chess, 2003). This section shows how the DARE functionalities performed by the various components in Figure 5.1 are mapped to MAPE activities (see Table 5.2).

In this research, the knowledge (K) in MAPE-K in the DARE framework is completely decentralized as indicated below:

- *Software architecture*. We assume that the software system starts with the software architecture not known at any node. DeSARM on every node then handles discovery of the current software architecture using gossiping techniques in a decentralized fashion.
- *Configuration Map*. None of the nodes have a complete view of the configuration of the software system. Instead, the configuration map is distributed using a distributed hash table.
- *Message-Based Transactions*. None of the nodes have a complete view of the transactions exchanged between application components. Instead, knowledge of these transactions is distributed among multiple RACs such that each RAC maintains only transactions to its component, as shown previously in chapters 3 and 4.

MAPE's monitoring activity is carried out by DeSARM, which is responsible for monitoring and suspecting node failures as part of the gossip exchanges between nodes. When DeSARM suspects a node failure, it notifies the FAM.

The analysis phase of the MAPE loop is performed by the FAM, which localizes recovery to the subnet that contains the suspected node and appoints one of the nodes in this subnet to coordinate recovery of the failed node. The FAM hosted by the recovery coordination node then proceeds with the analysis activity by pinging the suspected node to confirm the failure of this node. If this FAM does not receive a heartbeat reply from the suspected node within a certain time interval, it notifies the RAM at the same node of the failure. The analysis phase is also executed by the RACs, since RACs are responsible for analyzing transactions that were interrupted by failure.

MAPE's planning phase is executed by the RAM since it is responsible for determining the plan for adapting the current architecture and recovering every component deployed to a failed node. The RAM relies on (1) DeSARM for obtaining an up-to-date view of the software architecture and (2) the CM for obtaining the current configuration map. By using information from these two services, the RAM can proceed with planning adaptation or recovery as discussed in section 5.3. The planning phase is also executed by the RACs, since RACs are responsible for determining recovery actions required to recover any transactions that were interrupted due to failures.

The execution activity of the MAPE loop is performed by the RAM, which executes a reconfiguration template for creating failed components on a different node, connecting the recovered component with other components (as defined in the architecture obtained from DeSARM), and then activating the recovered component and its RAC so that they resume normal execution. The execution phase is also executed by the RACs, which are responsible for executing recovery actions required to recover any transactions that were interrupted due to failures.

Table 5.2 Mapping DARE functionalities to MAPE activities.

Monitoring	• DeSARM monitors and suspects node failure through lack of gossip messages from these nodes. • DeSARM activates FAM when it suspects a node failure.
Analysis	• FAM localizes recovery to a particular subnet and then appoints the recovery node to be the node with lowest IP address in the subnet. • FAM at the recovery node asserts failure of suspected node by pinging suspected node and then notifying RAM if no heartbeat message is received from the suspected node. • RAC stops sending messages to failed component and analyzes failed transactions.
Planning	• RAM determines components hosted by failed node. • RAM determines input and recipient RACs of each failed component. • RAM activates transaction recovery at input RACs of failed components. • RAM determines the recovery plan to recover each failed component. • RAC plans to recover failed transactions.
Execution	• RAM executes recovery template to: 　○ instantiate the failed component at a healthy node 　○ connect recovered component with neighbor components 　○ activate recovered component and recovery connectors to resume execution

	• RAC resumes sending messages, including recovered messages, to recovered component.

5.5 Component Failure Recovery

The previous sections described node failure recovery in which every component hosted by a failed node are recovered on a healthy node. This section describes how DARE can be extended to recover components that fail independently of the node that host them.

To address component failure recovery, the DARE framework is extended as follows:

1. In this research, every application component establishes a connection with DeSARM (located at the same node as the application component) in order for the component to exchange messages with other application components. Thus, when an application component fails, DeSARM detects that a connection is terminated unexpectedly and then sends Component Failed message (and supplies the identifier of the failed component) to the peer FAM on the same node.
2. When FAM receives a Component Failed message from DeSARM, it activates the peer RAM also on the same node. Note that component recovery does not require electing a recovery coordination node, since the failed component can simply be restarted on the same node.
3. When RAM is activated, it recovers the failed component as shown previously in section 5.3.1 but with two exceptions. First, the RAM selects the same node for

hosting the recovered component. Second, the RAM does not remove or add nodes to the configuration map simply because the system can recover to the same configuration it had before component failure.

6 DESIGN OF AN ASSISTANT RECOVERY AND ADAPTATION CONNECTOR FOR CLIENTS AND PRODUCERS

The previous chapters focused on the design of Recovery and Adaptation Connectors (RACs) that handle recovery and adaptation concerns of components that are coordinators, services, and consumers. However, these RACs do not handle recovery and adaptation of clients and producers. This chapter discusses the design of an Assistant Recovery and Adaptation Connector (ARAC) that can be used to handle adaptation and recovery concerns of clients and producers. The goal of the ARAC is to ensure that (1) responses to clients and requests and asynchronous messages sent by clients and producers can be recovered in case they are lost due to failure so that they are eventually received by the RAC and (2) clients and producers can be driven to a quiescent state in which they completed all transactions that they initiated and will not initiate any new transactions with other components. To increase its usability, the ARAC is designed so that it is applicable for both clients and producers in the following patterns:

- Clients in the synchronous message communication with reply, asynchronous message communication with callback, brokered communication, and SOA pattern.
- Producers in the unidirectional asynchronous message communication and the bidirectional asynchronous message communication.

To facilitate recovery of clients and producers, the ARAC maintains a log of the messages it sends and receives so that these messages can be recovered in case of failure. Recovery of connectors using message logs is described in the next chapter. This chapter discusses the design of the ARAC.

Each client or producer is associated with an ARAC as follows:

- When the ARAC receives a request from the client (Figure 6.1), it logs the request and then forwards the request to the service RAC. When the ARAC receives a service response, it logs the response, forwards the response to the client, and then sends an ACK message to the service RAC, so that the service RAC removes messages of this transaction from its queues.
- When the ARAC receives a unidirectional message from the producer (Figure 6.2), it logs the message and then forwards it to the consumer RAC. Eventually, the ARAC receives an ACK message from the consumer RAC indicating receipt of the producer message.

Since the ARAC is designed so that it is applicable for both clients and producers, the remainder of this chapter uses the term sender to refer to both clients and producers (i.e. a sender can send synchronous requests, asynchronous messages, or both type of messages). Furthermore, it is assumed that the ARAC does not fail. The remainder of this chapter describes the design of the ARAC and how it handles adaptation and recovery concerns of senders.

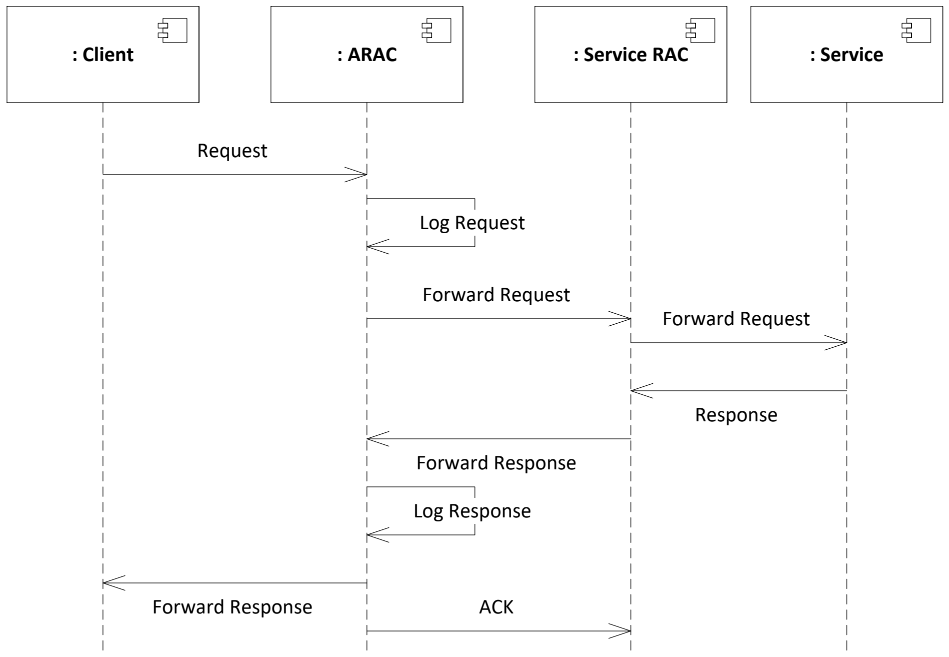


Figure 6.1 Behavior of ARAC during normal execution for transactions that comprise both request and response messages

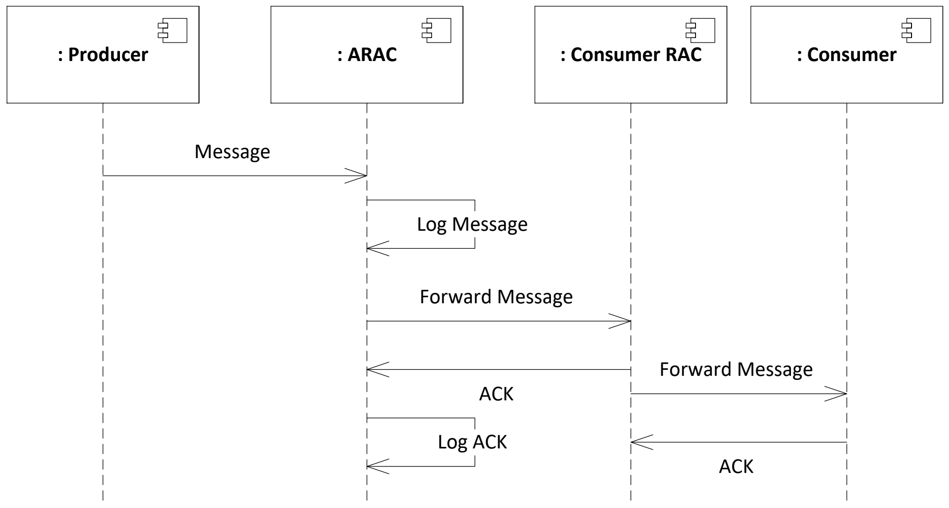


Figure 6.2 Behavior of ARAC during normal execution for transactions that comprise unidirectional messages

6.1 Design of the ARAC State Machine

The ARAC has the same structure as the RAC shown previously in Figure 3.1. However, the ARAC executes the state machine shown in Figure 6.3. This state machine is similar to the state machines shown previously in chapters 3 and 4 for handling services (c.f. Figure 3.3) and consumers (c.f. fig. 4.3). The differences are described in this chapter.

6.1.1 Normal Execution

During normal execution, when the ARAC receives a unidirectional message or a request while it is in the Waiting for Message state (Figure 6.3), the ARAC increments the transaction count, forwards this message to its destination, and then transitions to the Processing state.

While in the Processing state, the ARAC reacts to the various events as follows:

- If the ARAC receives an intermediate or final request, then the ARAC forwards this request to the service RAC.
- If the ARAC receives an asynchronous message, then the ARAC increments the transaction count and forwards the message to the service RAC.
- If the ARAC receives the first or an intermediate response, the ARAC forwards the response to the sender.
- If the ARAC receives a final response, the ARAC forwards the response to the sender, decrements the transaction count, and sends ACK to the service RAC. If the sender is not engaged in anymore transactions, the ARAC transitions to the Waiting for Message state.

- If the ARAC receives an ACK event from the service RAC indicating receipt of a previously sent asynchronous message, the ARAC decrements the transaction count. Furthermore, if the sender is not engaged in anymore transactions the ARAC transitions to the Waiting for Message state.

6.1.2 Adaptation of Sender

A sender can be adapted after it has received a response to every transaction it has initiated and must cease sending any asynchronous messages to the consumer while it is being dynamically adapted. The sender can resume sending messages only after it has been activated. Based on this, when the ARAC receives the Passivate command while it is in the Waiting for Message state (Figure 6.3), then the ARAC transitions immediately to the Quiescent state. Otherwise, if the ARAC receives the Passivate command while it is in the Processing state, then the ARAC transitions to the Passivating state and remains in this state until all currently active transactions are completed. While in the Passivating state, the ARAC holds all messages that initiate new transactions with the service in the Service Pending Queue. When all active transactions are completed, the ARAC transitions to the Quiescent state in which the ARAC also holds any new messages it receives from the sender in the Service Pending Queue. When the ARAC is reactivated after dynamic adaptation is completed, it resumes processing all held messages in the Service Pending Queue.

6.1.3 Recovery of Sender

Consider the case of the sender failure. It is assumed that when the sender recovers, the recovered sender must process the last response available at the ARAC to recover its state (Danilecki et al., 2013). Thus, when the sender fails, the Recovery and

Adaptation Manager (RAM) must recover another instance of the sender and notify the ARAC of sender failure as follows:

- The Recovery and Adaptation Manager (RAM) determines the RAC of the failed component as described in chapter 5. If no RAC is found for the failed component, then the RAM determines the ARAC of this component by looking up the architecture for any component that receives input messages (i.e. synchronous requests or asynchronous messages) from the failed component.
- The RAM notifies the ARAC of sender failure.
- The RAM recovers another instance of the sender and then connects the recovered sender with the ARAC.
- The RAM activates the sender and reactivates the ARAC.

As a result of notifying the ARAC of sender failure, the ARAC transitions to the Recovering state in which it recovers the last response it receives from the service using the following recovery actions:

- If the Response Forwarding Queue contains a service response, then the sender failed before the ARAC has forwarded this response to the sender. Therefore, the ARAC forwards this service response to the sender when it is recovered.
- If the Response Forwarding Queue is empty, then the ARAC recovers the last response it forwarded to the sender by moving this response from the Response Recovery Queue to the Response Forwarding Queue.

After the sender is recovered and the ARAC is reactivated, the ARAC forwards the response queued in the Response Forwarding Queue to the recovered sender.

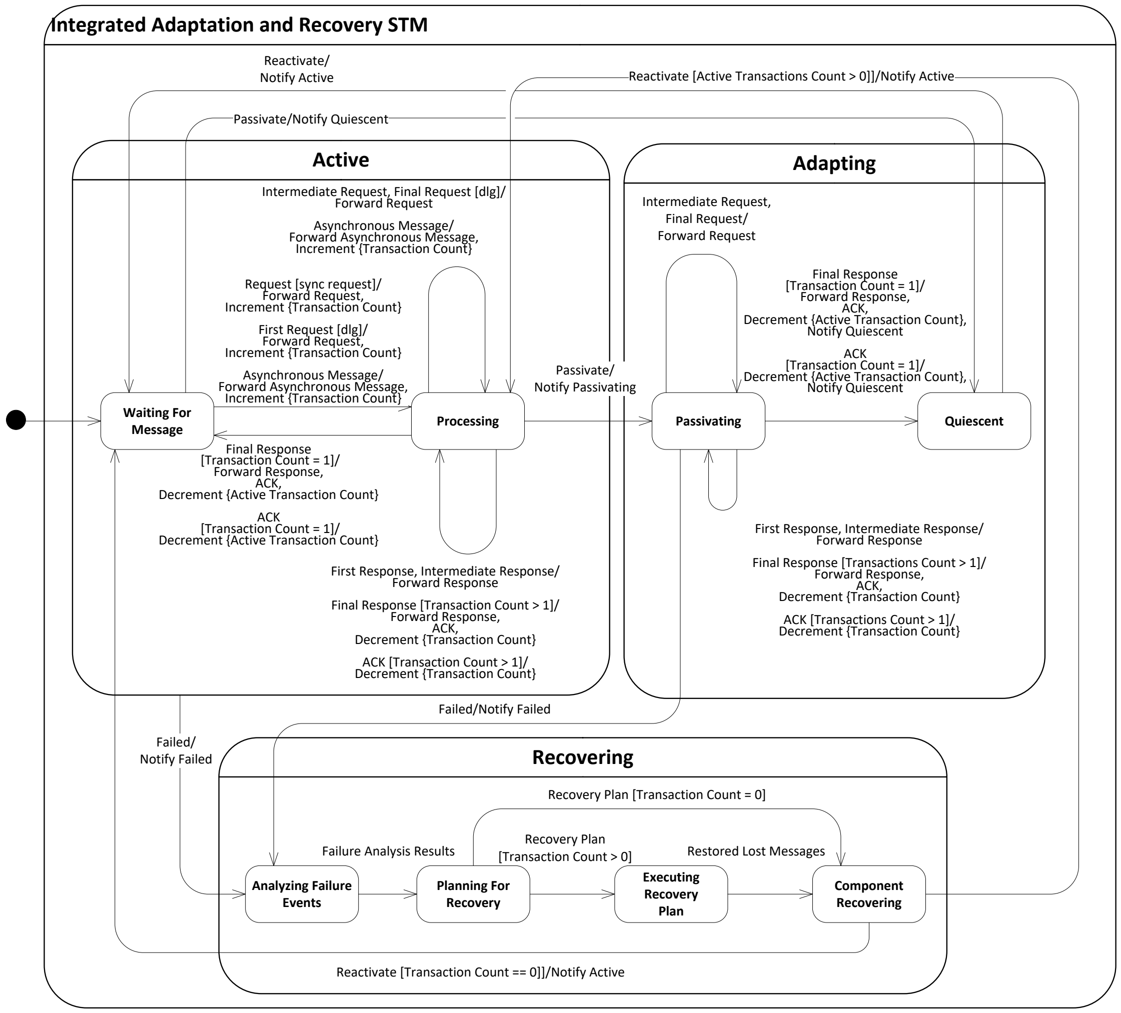

Integrated Adaptation and Recovery STM
Reactivate/ Notify Active
Reactivate [Active Transactions Count > 0]]/Notify Active
Passivate/Notify Quiescent
Active
Adapting
Intermediate Request, Final Request [dlg]/ Forward Request
Asynchronous Message/ Forward Asynchronous Message, Increment {Transaction Count}
Request [sync request]/ Forward Request, Increment {Transaction Count}
First Request [dlg]/ Forward Request, Increment {Transaction Count}
Asynchronous Message/ Forward Asynchronous Message, Increment {Transaction Count}
Waiting For Message
Processing
Final Response [Transaction Count = 1]/ Forward Response, ACK, Decrement {Active Transaction Count}
ACK [Transaction Count = 1]/ Decrement {Active Transaction Count}
First Response, Intermediate Response/ Forward Response
Final Response [Transaction Count > 1]/ Forward Response, ACK, Decrement {Transaction Count}
ACK [Transaction Count > 1]/ Decrement {Transaction Count}
Passivate/ Notify Passivating
Intermediate Request, Final Request/ Forward Request
Final Response [Transaction Count = 1]/ Forward Response, ACK, Decrement {Active Transaction Count}, Notify Quiescent
ACK [Transaction Count = 1]/ Decrement {Active Transaction Count}, Notify Quiescent
Passivating
Quiescent
First Response, Intermediate Response/ Forward Response
Final Response [Transactions Count > 1]/ Forward Response, ACK, Decrement {Transaction Count}
ACK [Transactions Count > 1]/ Decrement {Transaction Count}
Failed/Notify Failed
Failed/ Notify Failed
Recovering
Recovery Plan [Transaction Count = 0]
Failure Analysis Results
Recovery Plan [Transaction Count > 0]
Restored Lost Messages
Analyzing Failure Events
Planning For Recovery
Executing Recovery Plan
Component Recovering
Reactivate [Transaction Count == 0]]/Notify Active

Figure 6.3 State machine executed by the ARAC

7 CONNECTOR RECOVERY

The previous chapters assumed that the recovery and adaptation connector (RAC) does not fail. As a preliminary research effort, this chapter investigates relaxing this assumption by considering how the RAC can be recovered after a run-time failure. To handle recovery of the RAC, the approach involves storing the messages that the RAC receives into a log so that the RAC can reconstruct the state of its queues by replaying messages from the log (Tanenbaum and Steen, 2006). To ensure recoverability of the RAC's log, the log is replicated such that each replica of the log is stored in a different node than other replicas of the same log.

This chapter discusses the approach for recovering RACs using logs. Section 7.1 describes the message logging approach used by the RAC. Section 7.2 discusses how messages are logged during normal execution. Section 7.3 discusses how messages can be replayed to reconstruct the state of the RAC queues after a runtime failure. Section 7.4 discusses how messages that were lost while the RAC in the failed state can be recovered.

7.1 Message Logging Approach

As there exists many approaches for message logging, this research considers using the *pessimistic message logging* in which the RAC synchronously logs every incoming application message to stable storage before it processes the message.

Pessimistic message logging is used in this research for the following reasons (Elnozahy et al., 2002):

- Recovery using pessimistic message logging is simple and confined to the recovered component (i.e. RAC).
- A RAC that recovers from a failure does not require any application component to rollback its state to a previous state due to the failure of the RAC. This is a highly desirable property since components in some application domains, such as SOAs, are highly autonomous and cannot be forced to rollback their states.

In pessimistic message logging, a component must log information about every application message it receives before processing that message. Thus in this research, all input and output messages received by the RAC from application components must be logged by the RAC. When the RAC recovers from a run-time failure, the recovered RAC replays messages from its log so that it can reconstruct the state of its queues, as will be explained in the next sections.

7.2 Behavior During Normal Execution

As explained in the previous section, the RAC must store the messages it receives in a log so that these messages can be replayed during recovery time in order to reconstruct the state of the RAC queues. To accomplish this, an input and an output stubs are used to facilitate message logging at the RAC (see Figure 7.1). The goal of these stubs is to separate message logging and replaying concerns from recovery and adaptation concerns carried out by the RAC. The input stub handles requests from clients to the RAC and responses from the RAC to clients . On the other hand, the output stub handles requests sent from the RAC to the service and also responses from the service to the RAC. When it receives an input message, the input stub logs and then forwards the input message to the RAC. Similarly, when the output stub receives an output response, it logs and then forwards the response to the RAC.

This section discusses the behavior of the stubs during normal execution for message logging. The next section describes how these stubs replay messages after a RAC run-time failure to recover the state of the RAC queues.

During normal execution, the input and output stubs update the RAC log as follows:

1. When the input stub of the RAC receives a client request (message 2 (referred to as m2)), it logs the client request (m3) and then forwards this request to the RAC (m5). In this research, the RAC log is stored in a distributed hash table (DHT) that supports replication of its entries to ensure recoverability of the RAC log in case of failure.
2. The RAC processes the client request and then forwards this request to the service via the output stub (m6).
3. The output stub forwards the client request to the service (m7).
4. The service sends the service response to the RAC via the output stub (m8).
5. The output stub logs the service response (m9) and then forwards the service response to the RAC (m11).

6. The RAC processes the service response and then forwards this response to the requesting client via the input stub (m12).
7. The input stub forwards the service response to the client (m13).

Thus, this approach ensures that all application messages that the RAC processes are maintained in a message log.

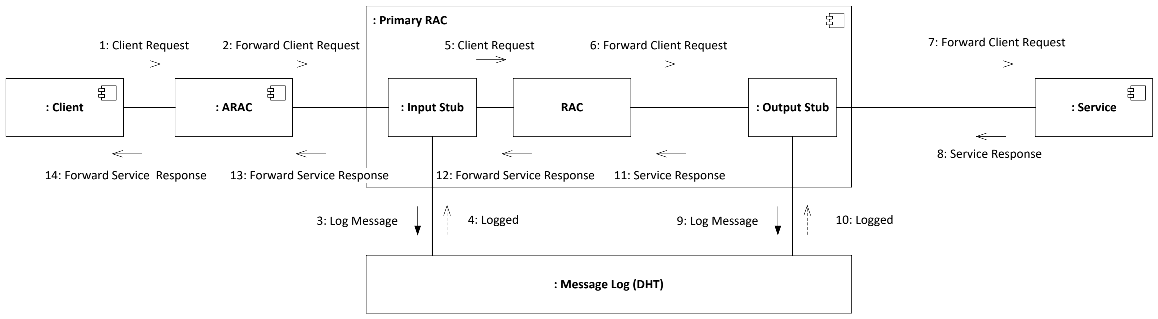


Figure 7.1 Message logging during normal execution.

7.3 Reconstructing the RAC State After a Runtime Failure

When the RAC fails, the Recovery and Adaptation Manager (RAM) recovers another instance of the RAC and notifies the ARAC of the new location of the RAC. As a result, the input and output stubs of this newly recovered instance of the RAC must first recover the state of the RAC's queues by replaying messages from the log as follows:

1. The input and output stubs retrieve the RAC's log from the DHT. Note the access to the log by these stubs is synchronized.

2. The input stub iterates over the messages in the log. For each client request in the log, the input stub marks the request as *replayed* and then forwards this request to the RAC.
3. The RAC processes the client request and eventually forwards this request to the output stub.
4. When the output stub receives (from the RAC) a client request that is marked as *replayed*, it checks the log to see whether it contains a corresponding response to this request by using the message identifier. If the log contains such a response, then the output stub marks the response as *replayed* and sends this response to the RAC. This ensures that a service response is not replayed by the output stub to the RAC unless the RAC has first processed the client request. Note that the output stub does not forward to the service any requests that are marked as *replayed* to avoid sending duplicate requests to the service.
5. The RAC processes the service response and eventually forwards this response to the input stub.
6. When the input stub receives (from the RAC) a service response that is marked as *replayed*, it discards this response to avoid sending duplicate responses to clients.

At the end of this process, the RAC has processed all replayed client requests and service responses. As a result, the state of the RAC queues are reconstructed.

7.4 Recovery of Lost Messages

Failure of the RAC can cause some transactions to be interrupted at different statuses. Thus, after the RAC reconstructs its state by replaying messages from its log, the recovered RAC must execute certain recovery actions to recover any interrupted transactions based on the status of the transaction. Note that the recovery actions executed by the recovered RAC for a transaction are based on the pattern of that transaction (see chapters 3 and 4). The reminder of this section shows an example of a non-distributed transaction (see section 3.1 in chapter 3) that fails due to failure of the RAC. Figure 7.2 depicts an example of a fault-tree model (Ericson, 1999) with the possible interruption points of a non-distributed transaction. A description of each interruption point in this tree and the corresponding recovery action are provided as follows:

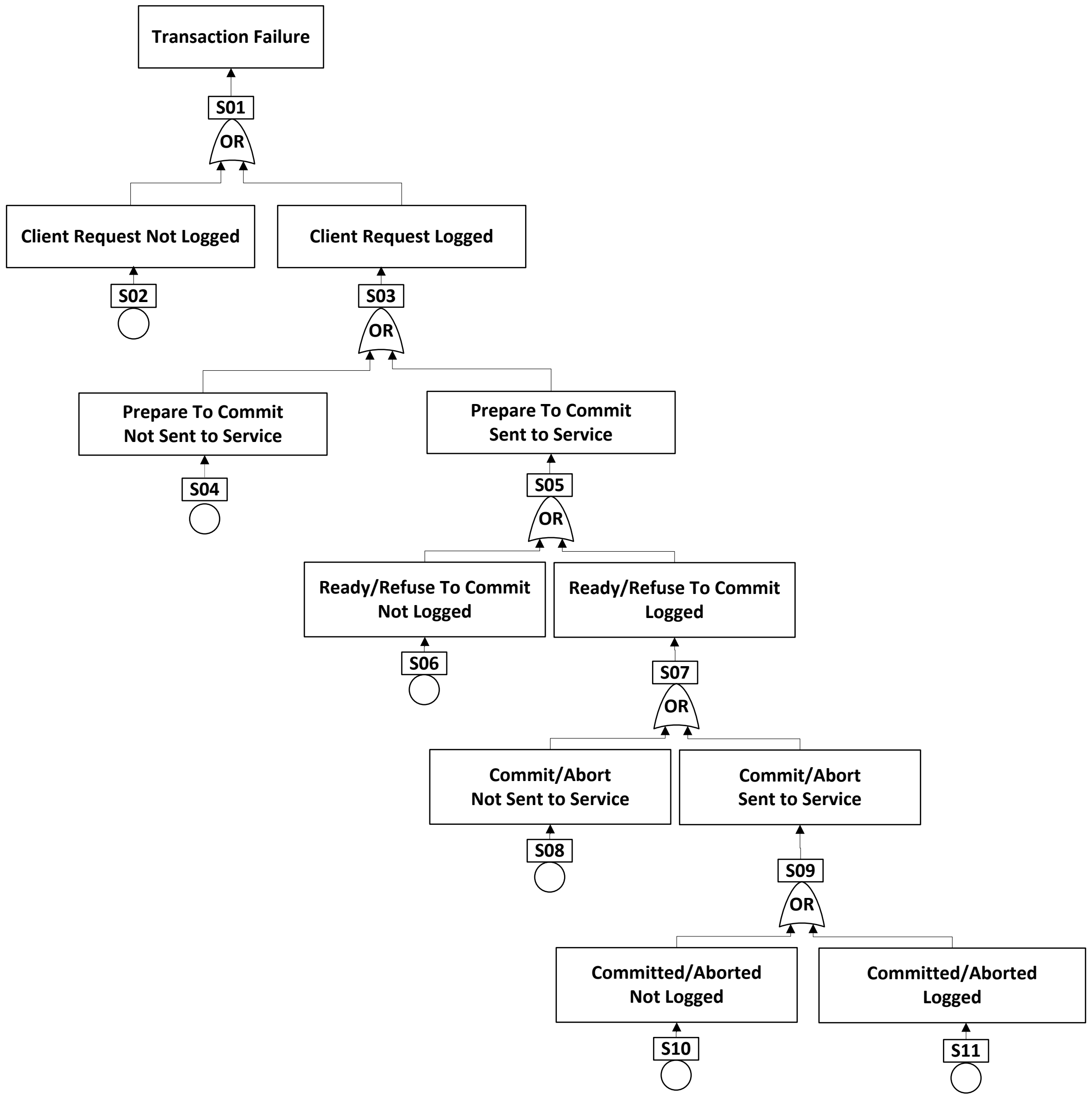

Transaction Failure
S01
OR
Client Request Not Logged
Client Request Logged
S02
S03
OR
Prepare To Commit Not Sent to Service
Prepare To Commit Sent to Service
S04
S05
OR
Ready/Refuse To Commit Not Logged
Ready/Refuse To Commit Logged
S06
S07
OR
Commit/Abort Not Sent to Service
Commit/Abort Sent to Service
S08
S09
OR
Committed/Aborted Not Logged
Committed/Aborted Logged
S10
S11

Figure 7.2 Transaction interruption points depicted in a fault-tree model.

1. The RAC could have failed after receiving a client request but before logging this request (i.e. interruption point S02 in Figure 7.2). This case results in a lost client request that is not forwarded to service. To solve this issue, the ARAC is notified of the RAC failure by the Recovery and Adaptation Manager (RAM). As a result, the

ARAC resends the client request to the RAC when the RAC is recovered, since the ARAC has not received a response for the request it forwarded to the RAC.

2. The RAC could have failed after logging the client request but before sending the Prepare To Commit message to the service (i.e. interruption point S04). This case results in the recovered RAC waiting indefinitely for a service response to this request, since the transaction has not been initiated with the service. To recover from this case, the recovered RAC (1) instructs the service to abort all transactions that are in the *preparing* state and then (2) restarts these transactions with the service. The rationale for these recovery actions is that the recovered RAC is uncertain whether the service has received these transactions or these transactions were lost due to failure. To avoid sending duplicate requests to the service, the recovered RAC instructs the service to abort these transactions and then restarts them with the service.
3. The RAC could have failed after receiving the Ready To Commit (or Refuse To Commit) message from the service but before logging this response (i.e. interruption point S06 in Figure 7.2). This case results in a transaction that stays indefinitely in the *prepared* (or *refused*) state. Although this case is indistinguishable from case 2 from the point of view of the recovered RAC, the recovered RAC can recover this transaction using the same recovery actions used in the previous case (i.e. instructing the service to abort the transaction and then restarting this transaction with the service).
4. The RAC could have failed before sending the Commit (or Abort) message to the service (interruption point S08). To recover from this case, the recovered RAC

instructs the service to either commit or abort the transaction according to the service response that this RAC has received from the service before it failed.

5. The RAC could have failed after receiving the Committed (or Aborted) message from the service but before logging this response (i.e. interruption point S10 in fig. 7.3).To recover from this case, the RAC resends the Commit (or abort) message to the service.

8 DESIGN OF A REUSABLE RECOVERY AND ADAPTATION CONNECTOR

Chapters 3 and 4 discussed the design of different types of the Recovery and Adaptation Connector (RAC) for different architectural patterns. For instance, the design of the RAC for handling adaptation and recovery concerns of stateless services in service-oriented architectures (SOAs) is different from the design of the RAC used to handle stateful services which are both different from the design of the RAC for handling consumers in asynchronous patterns. Given these variations in the RACs, it would be beneficial to apply reuse concepts to unify the RAC design in order to increase its usability.

This chapter investigates this problem by showing how the software product line (SPL) technology can be used to design a reusable RAC that can be tailored to (1) generate different types of RAC as well as (2) generate RACs that can handle integration of multiple patterns such as a RAC that can handle both synchronous message communication with reply and asynchronous message communication.

This chapter is organized as follows. Section 8.1 discusses how variability in the different types of the RAC can be captured using a feature model. Section 8.2 discusses the impact of features in the feature model on the RAC design. Section 8.3 describes the feature-based state machine executed by Connector Control of the reusable RAC.

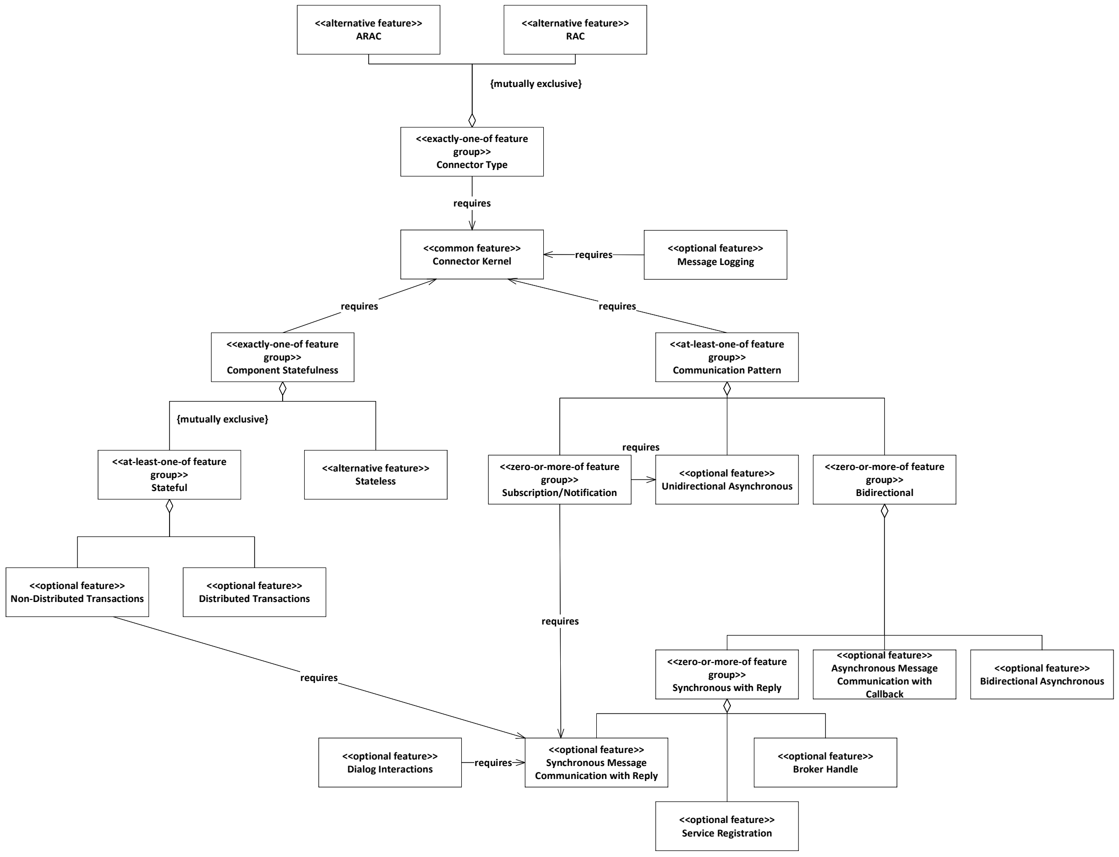
8.1 Capturing Variability in the RAC using a Feature Model

Before a reusable RAC can be designed, commonality and variability among the different RAC types must be captured using a feature model. In SPL technology, feature models are used to depict (1) deviations, in terms of features, between different products of the same product line and (2) the dependency relationships between these features (Clements and Northrop, 2001).

In this research, the different RAC types are considered products of the same product line, and the feature model describes how the different types of the RAC vary among each other. The approach used in this research for designing the feature model and the reusable RAC is based on the PLUS method (Gomaa, 2004), which has been used previously in conjunction with adaptation connectors to adapt from one member of a SPL to another (Gomaa and Hashimoto, 2011).

The PLUS method uses the Unified Modeling Language (UML) metaclass notation for representing features and UML stereotypes for categorizing features. Furthermore, features can be grouped together based on constraints on their selection for a given RAC type, and one feature may require another feature for its existence.

Figure 8.1 shows the feature model for the reusable RAC that is constructed by analyzing the various RAC designs shown previously in chapters 3, 4, and 6. The description of these features is given next.



<<alternative feature>> ARAC
<<alternative feature>> RAC
{mutually exclusive}
<<exactly-one-of feature group>> Connector Type
requires
<<common feature>> Connector Kernel
requires
<<optional feature>> Message Logging
requires
requires
<<exactly-one-of feature group>> Component Statefulness
<<at-least-one-of feature group>> Communication Pattern
{mutually exclusive}
<<at-least-one-of feature group>> Stateful
<<alternative feature>> Stateless
<<zero-or-more-of feature group>> Subscription/Notification
requires
<<optional feature>> Unidirectional Asynchronous
<<zero-or-more-of feature group>> Bidirectional
<<optional feature>> Non-Distributed Transactions
<<optional feature>> Distributed Transactions
requires
requires
<<zero-or-more-of feature group>> Synchronous with Reply
<<optional feature>> Asynchronous Message Communication with Callback
<<optional feature>> Bidirectional Asynchronous
<<optional feature>> Dialog Interactions
requires
<<optional feature>> Synchronous Message Communication with Reply
<<optional feature>> Broker Handle
<<optional feature>> Service Registration

Figure 8.1 Feature model for the reusable RAC.

Common features. Common features are features that must be supported by all types of RAC, such as the ability for a RAC to receive input messages and then to forward these messages to its component.

Exactly-one-of feature groups. Exactly-one-of feature groups represent mutually exclusive features that cannot coexist in a single RAC. In Figure 8.1, the *connector type* feature group represents the type of the connector which can be either a RAC (see chapters 3 and 4) or an ARAC (see chapter 6). Another example of an exactly-one-of feature group is the *Component Statefulness*. In particular, the component handled by a RAC can be either *Stateless* or *Stateful*. The design of the RAC for stateless components is described in section 3.1. However, as described previously in the design of the RAC for stateful components (see section 3.2), the RAC must forward messages to a stateful component using the Two-Phase Commit (2PC) protocol. Since using the 2PC protocol with stateless components is unnecessary, because it increases complexity and is less efficient, the RAC must not use the 2PC protocol with stateless components. As a result, the *stateless* and *stateful* features are treated as mutually exclusive features in Figure 8.1.

At-least-one-of feature groups. At-least-one-of feature groups represent groups from which at least one feature must be selected. In Figure 8.1, the *stateful* feature group represents the type of stateful transactions that the reusable RAC can manage. In particular, a transaction can be non-distributed (c.f. section 3.2) or distributed (c.f. section 3.3.5). Another example of the at-least-one-of feature group is the *Communication Pattern* feature group which represents the communication patterns that the RAC participates in. In particular, a RAC can manage either *Unidirectional Asynchronous* communication (c.f. section 4.1), *Bidirectional* communication (c.f. sections 3.1, 3.3, and 4.3.1), *Subscription/Notification* (c.f. section 4.3.2) or any combinations of these types of communication.

Zero-or-more-of feature groups. Zero-or-more-of feature groups represent groups from which zero or more features can be selected. In Figure 8.1, the *Bidirectional* feature group represents possible bidirectional communication patterns that the reusable RAC can manage. In particular, the reusable RAC can handle bidirectional communications that involve *Synchronous with Reply* (c.f. section 3.1), *Asynchronous with Callback* (c.f. section 3.3.1), *Bidirectional Asynchronous* (c.f. 4.3.1), or any combinations of these types of bidirectional communications. The *Synchronous with Reply* feature itself is a zero-or-more-of group since this type of communication can involve several communication patterns including *Broker Handle* (c.f. section 3.3.3), *Service Registration* (c.f. section 3.3.2), or the *Synchronous Message Communication with Reply* (c.f. section 3.1).

Optional features. Optional features represent features that are supported by some (but not all) types of RAC. In Figure 8.1, in case of *Synchronous Communication with Reply*, transactions can optionally comprise *Dialog Interactions* between components. Note that in the synchronous communication with reply, a client sends a request to the service and then blocks waiting for the service response. On the other hand, dialog interactions are used when the client needs to send multiple requests synchronously to the service such that the next request in the dialog depends on the response of the previous request. As a result, the entire dialog is considered as a compound transaction. Another example of an optional feature is the *Message Logging* feature which enables the RAC to log the messages it receives (see chapter 7).

Given the feature model in Figure 8.1, the goal is to design a product line reusable RAC that satisfies this model, such that this reusable RAC can be tailored based on feature selection to generate different types of the RAC.

8.2 **Feature/Component Table**

In order to understand the impact of each feature in Figure 8.1 on the design of the reusable RAC, the reuse category of each component in the RAC must be analyzed. Based on this analysis, a feature/component table is constructed (see Table 8.1) to map the impact of each feature on each component of the reusable RAC. Then from this table, a reuse stereotype is assigned to components (see Figure 8.2). The remainder of this section discusses the impact of features on each RAC component.

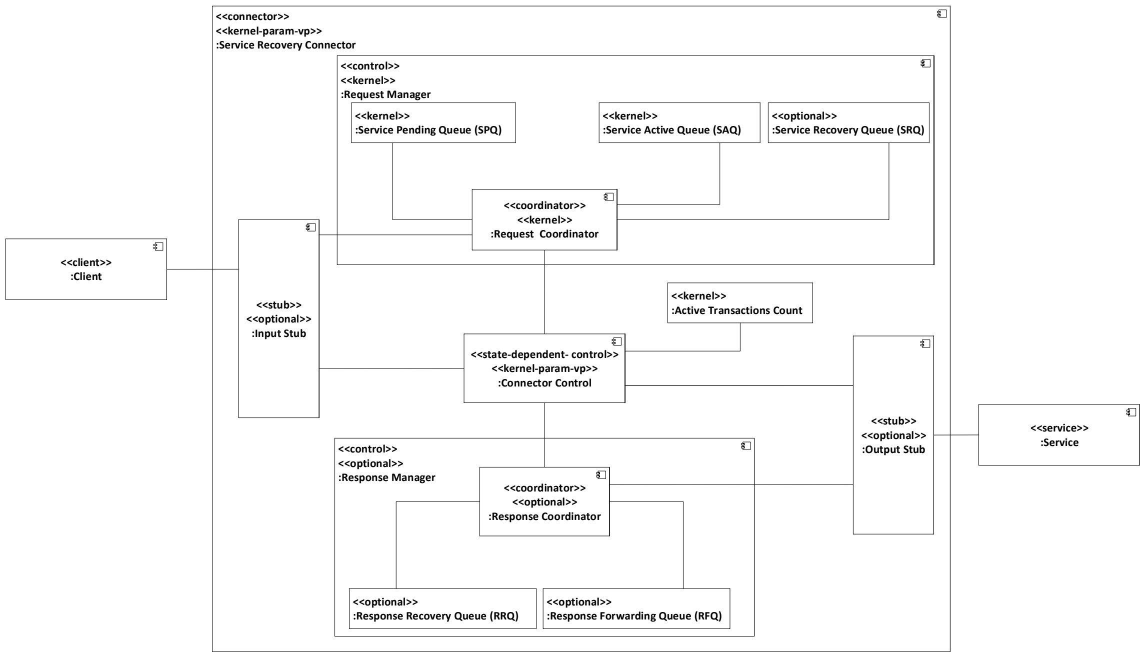


Figure 8.2 Design of the reusable RAC with reuse stereotypes.

Request Manager. The Request Manager is responsible for managing input messages, including synchronous requests and asynchronous messages, to the RAC. Since every

RAC must receive input messages to its component, the Request Manager is a *kernel* composite component (see Figure 8.2) that must always be supported by all RAC types.

Request Coordinator. The Request Coordinator is responsible for (1) sending input messages to Connector Control for further processing as well as (2) managing the various queues in the Request Manager. Since a RAC must have an input message queue and input messages must always be forwarded to Connector Control, then the Request Coordinator is a *kernel* component.

Connector Control. Connector Control is responsible for forwarding messages to its application component and handling adaptation and recovery concerns of this component. Since forwarding messages must be supported by all RAC types, then Connector Control is a *kernel* component. However, as described previously in chapters 3 and 4, the behavior of Connector Control for handling adaptation and recovery concerns differs based on the *Statefulness* of the component, the selected *Communication Patterns,* and whether *Dialog Interactions* and *Distributed Transactions* features are supported. Therefore, Connector Control is a parameterized component whose behavior can change based on the selection of these features (see Table 8.1). Thus, these feature conditions are used to tailor the reusable state machine executed by Connector Control, as will be discussed in section 8.3.

Response Manager. The Response Manager is responsible for receiving and maintaining output responses. As shown previously in chapter 4, RACs in asynchronous patterns do not receive responses from their components. Therefore, the Response Manager is an optional composite component that is only selected for *Synchronous with Reply* communication and/or *Asynchronous Communication with Callback Handle*.

Response Coordinator. The Response Coordinator is responsible for managing queues that maintain service responses. As shown previously in chapter 4, RACs in asynchronous patterns do not receive responses from their components. Therefore, the Response Coordinator is an optional component that is only selected for *Synchronous with Reply* communication and/or *Asynchronous Communication with Callback Handle*.

Input and Output Stubs. The Input and Output Stubs are responsible for message logging and message replaying as described previously in chapter 7. These stubs are optional objects that are only selected if the *Message Logging* feature is selected.

Table 8.1 Feature/component table.

Feature Name	**Feature Category**	**Component Name**	**Reuse Category**	**Feature Condition**
RAC Kernel	Common	Request Manager Request Coordinator	Kernel Kernel	

		Connector Control	Kernel, Parameterized	
Stateless	Alternative	Connector Control Response Manager Response Coordinator	Kernel, Parameterized Optional Optional	stateless
• Synchronous with Reply • Async. Comm. with Callback	Optional	Connector Control Response Manager Response Coordinator	Kernel, Parameterized Optional Optional	sync request
• Unidirectional Asynchronous • Bi-directional Asynchronous	Optional	Connector Control	Kernel, Parameterized	async
Dialogs	Optional	Connector Control Response Manager Response Coordinator	Kernel, Parameterized Optional Optional	dlg

Distributed Transactions	Alternative	Connector Control	Kernel, Parameterized	distributed tr
Non Distributed Transactions	Alternative	Connector Control	Kernel, Parameterized	Non-distributed tr
RAC	Alternative	Connector Control	Kernel, Parameterized	RAC
ARAC	Alternative	Connector Control	Kernel, Parameterized	ARAC
Message Logging	Optional	Input Stub Output Stub	Optional Optional	

8.3 **Feature-Based Connector Control State Machine**

As described in section 8.2, Connector Control (CC) is a kernel, parameterized component whose behavior changes based on feature selection. The goal of CC is to handle adaptation and recovery concerns of its component. In order to manage the complexity of integrating multiple adaptation and recovery state machines into CC, a reusable state machine is constructed from which different types of RAC can be generated. The approach involves mapping patterns to features, and then augmenting transitions in the reusable state machine with feature conditions such that transitions can be enabled or disabled based on feature selection as follows:

- The design of the RAC for handling adaptation and recovery concerns of *stateless* services (c.f. fig. 3.3 in chapter 3) is mapped to the *stateless* feature. As a result, transitions with pattern-based events in this state machine are augmented with the feature conditions *RAC* and *stateless*.

- The design of the RAC for handling adaptation and recovery concerns of *stateful* services that are involved in *non-distributed* transactions (c.f. fig. 3.6 in chapter 3) is mapped to the *non-distributed transactions* feature. As a result, transitions with pattern-based events in this state machine are augmented with the feature condition *RAC* AND *non-distributed tr*. Note that this pattern may also involve transactions that involve synchronous request with replay and/or dialog interactions. Therefore, the state machine in fig. 3.6 in chapter 3 is augmented with the *RAC*, *sync request* and *dlg* conditions as well.

- The design of the RAC for handling adaptation and recovery concerns of *stateful* services that are involved in *distributed* transactions (c.f. fig. 3.14 in chapter 3) is mapped to the *distributed transactions* feature. As a result, transitions with pattern-based events in this state machine are augmented with the feature condition *RAC* and *distributed tr*.

- The design of the RAC for handling adaptation and recovery concerns of consumers in asynchronous patterns (c.f. fig. 3.3 in chapter 3) is mapped to the *Unidirectional* and *Bidirectional Asynchronous* features. As a result, transitions with pattern-based events in this state machine are augmented with the feature condition *RAC* and *async*.

- The design of the Assistant Recovery and Adaptation Connector (c.f. chapter 6) is mapped to the *ARAC* feature. As a result, transitions with pattern-based events in this state machine are augmented with the feature condition *ARAC*.

At this point, a reusable state machine is constructed for CC by integrating the pattern-based state machines such that every pattern-based transition in this reusable state machine has a guard with a feature condition. Thus, these transitions can be enabled or disabled using feature selection as follows:

- If the *stateless* feature is selected, then the *stateless* feature condition must be set for CC (see table 8.1), which enables all transitions of the integrated adaptation and recovery state machine used to handle *stateless* services.

- If the *distributed transaction* feature is selected, then the *distributed tr* feature condition must be set for CC (see table 8.1), which enables all transitions of the integrated adaptation and recovery state machine used to handle services involved in distributed transactions.

- If the *non-distributed transaction* feature is selected, then the *non-distributed tr* feature condition must be set for CC (see table 8.1), which enables all transitions of the integrated adaptation and recovery state machine used to handle services involved in non-distributed transactions.

- If either the *Unidirectional* or *Bidirectional Asynchronous* features is selected, then the *async* feature condition must be set for CC (see table 8.1), which enables all transitions of the integrated adaptation and recovery state machine used to handle consumers involved in asynchronous patterns.
- If the *ARAC* feature is selected, then the *ARAC* feature condition must be set for CC (see table 8.1), which enables all transitions of the integrated adaptation and recovery state machine executed by the ARAC.

In this research, every message contains metadata that is used by the reusable RAC to identify the pattern involved, including the communication type (e.g. asynchronous or synchronous with reply) as well as whether the transaction is distributed and whether it involves a dialog. Therefore, the reusable RAC is capable of setting feature conditions based on metadata of the messages it receives. However, the stateless and ARAC feature conditions must be set at deployment time of each RAC. Note that if the ARAC feature condition for ARAC is selected, then the reusable RAC is configured as an ARAC.

8.3.1 Example of RAC Generation

This section shows an example of tailoring CC based on feature selection to support multiple patterns as shown in section 8.3. In this example a RAC must be tailored for handling a stateful service that participates in both distributed and non-distributed transactions.

In this example, since the service is stateful and participates in both distributed and non-distributed transactions, then the *RAC, non-distributed transactions* and *distributed transactions* features must be selected. As a result of selecting these features, CC executes the state machine shown in Figure 8.3 which enables the transitions of the state machines of the two corresponding patterns for handling non-distributed transactions (c.f. fig. 3.6 in chapter 3) and for handling distributed transactions (see fig. 3.14 in chapter 3).

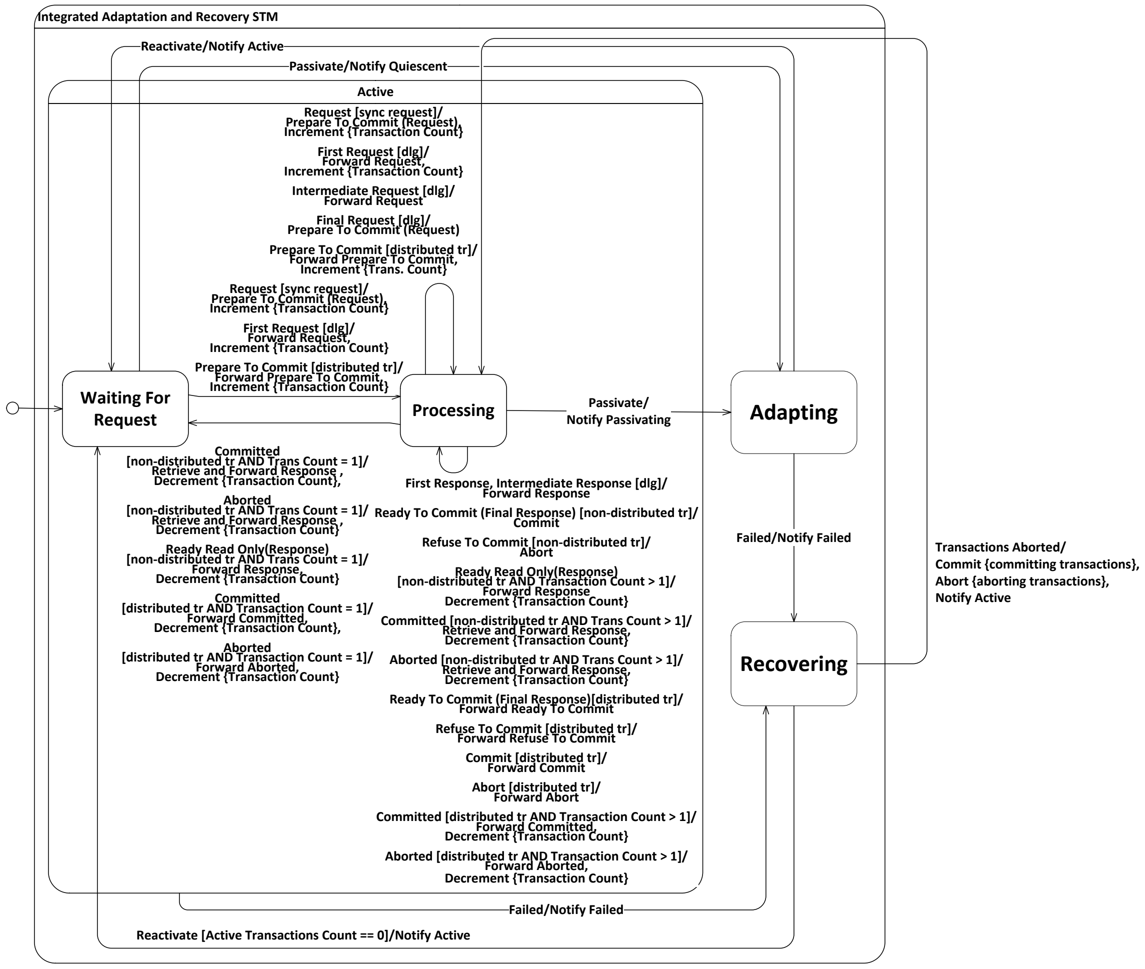

Integrated Adaptation and Recovery STM
Reactivate/Notify Active
Passivate/Notify Quiescent
Active
Request [sync request]/ Prepare To Commit (Request), Increment {Transaction Count}
First Request [dlg]/ Forward Request, Increment {Transaction Count}
Intermediate Request [dlg]/ Forward Request
Final Request [dlg]/ Prepare To Commit (Request)
Prepare To Commit [distributed tr]/ Forward Prepare To Commit, Increment {Trans. Count}
Request [sync request]/ Prepare To Commit (Request), Increment {Transaction Count}
First Request [dlg]/ Forward Request, Increment {Transaction Count}
Prepare To Commit [distributed tr]/ Forward Prepare To Commit, Increment {Transaction Count}
Waiting For Request
Processing
Passivate/ Notify Passivating
Adapting
Committed [non-distributed tr AND Trans Count = 1]/ Retrieve and Forward Response , Decrement {Transaction Count},
Aborted [non-distributed tr AND Trans Count = 1]/ Retrieve and Forward Response, Decrement {Transaction Count}
Ready Read Only(Response) [non-distributed tr AND Trans Count = 1]/ Forward Response, Decrement {Transaction Count}
Committed [distributed tr AND Transaction Count = 1]/ Forward Committed, Decrement {Transaction Count},
Aborted [distributed tr AND Transaction Count = 1]/ Forward Aborted, Decrement {Transaction Count}
First Response, Intermediate Response [dlg]/ Forward Response
Ready To Commit (Final Response) [non-distributed tr]/ Commit
Refuse To Commit [non-distributed tr]/ Abort
Ready Read Only(Response) [non-distributed tr AND Transaction Count > 1]/ Forward Response Decrement {Transaction Count}
Committed [non-distributed tr AND Trans Count > 1]/ Retrieve and Forward Response, Decrement {Transaction Count}
Aborted [non-distributed tr AND Trans Count > 1]/ Retrieve and Forward Response, Decrement {Transaction Count}
Ready To Commit (Final Response)[distributed tr]/ Forward Ready To Commit
Refuse To Commit [distributed tr]/ Forward Refuse To Commit
Commit [distributed tr]/ Forward Commit
Abort [distributed tr]/ Forward Abort
Committed [distributed tr AND Transaction Count > 1]/ Forward Committed, Decrement {Transaction Count}
Aborted [distributed tr AND Transaction Count > 1]/ Forward Aborted, Decrement {Transaction Count}
Failed/Notify Failed
Recovering
Transactions Aborted/ Commit {committing transactions}, Abort {aborting transactions}, Notify Active
Failed/Notify Failed
Reactivate [Active Transactions Count == 0]/Notify Active

Figure 8.3 CC state machine for handling integration of distributed and non-distributed transactions.

8.4 Possible Optimizations

Chapters 3 and 4 described how the RAC requires additional ACK messages for various patterns so that it can remove transaction messages from its queues. This section describes the number of additional messages required for each different type of RAC and possible optimizations to reduce this number (see Table 8.2). For all types of RACs, an additional ACK message is required at the end of the transaction so that the RAC can remove the transaction messages from its queues. In SOA-related patterns, these ACK messages can be piggybacked into the next transaction that the client initiates with the service. In the unidirectional asynchronous message communication, the consumer can periodically send a single ACK message to the consumer RAC for acknowledging multiple transactions. In case of the synchronous message communication with reply when the service has non-idempotent operations, there are two additional messages for each transaction that correspond to the second phase of the 2PC protocol. The committed/aborted response from the service to the service RAC can be piggybacked into the next response of another transaction (e.g. ready To Commit) that the service sends to the RAC. Note that in case the service does not receive any transactions from the RAC within a certain time interval, then the service has to send explicit committed/aborted messages to the RAC.

Table 8.2 Number of additional messages required by each type of RAC and possible optimizations

Pattern	**Number of Additional Messages Per Transaction**	**Possible Optimizations**
Synchronous Message Communication with reply (stateless service)	One additional ACK message is needed when transaction is completed so that the service RAC can remove transaction messages from its queues.	The ACK message can be piggybacked into the next transaction that the client initiates with the service.

Synchronous Message Communication with reply (stateful service with non-idempotent operations)	There are two cases to consider: • If the operation is read-only, then similar to the previous pattern, only one ACK is needed to indicate completion of transaction. • If the operation is write, then in addition to the ACK message, there are two extra messages needed for the second phase of 2PC.	ACK messages can be handled in the same way as the previous pattern. In case of write operations, the committed or aborted confirmation message from the service to the RAC can be piggybacked into the next response (e.g. readyToCommit) that the service sends to the service RAC. Thus, only one extra message is needed from the RAC to the service for instructing the service to either commit or abort the transaction. Note that if the service does not receive a new transaction from the RAC within a certain time interval, then the service must send explicit

		committed/aborted messages to the service RAC.
Distributed Transaction	*1+n* additional ACK messages are needed when transaction is completed where *n* is the number of services in the pattern.	ACK message can be piggybacked into the next transaction, similar to the previous two patterns.
Unidirectional Asynchronous Message Communication	One additional ACK message from consumer to consumer RAC is needed to indicate that the consumer is done with the producer message.	The consumer can periodically send one message to consumer RAC to acknowledge completion of multiple transactions (instead of a single transaction).

9 FORMAL PROPERTIES

This section defines formal recovery and adaptation properties and proofs that are ensured by both the DARE framework and Recovery and Adaptation Connectors (RACs). Section 9.1 defines several concepts formally, including software architecture, configuration map, architectural pattern and transactions which are used throughout this chapter. Section 9.2 defines properties related to the DARE framework. Section 9.3 defines properties related to the RACs in SOA patterns when services are stateless. Section 9.4 defines properties related to the RACs in SOA patterns when services are stateful. Section 9.5 defines properties related to the RACs in asynchronous patterns.

9.1 Definitions

This section formally defines several concepts that are used throughout this chapter, including software architectures, configuration maps, and architectural patterns and transactions.

9.1.1 Software Architecture

In this dissertation, a software architecture SA is defined as the set $SA = \{O, I\}$ where:

- O is the set of objects in the architecture such that an object can be either an application component or a Recovery and Adaptation Connector (RAC).

Formally, $O = \{o \mid o \in C \cup R\}$ such that $C = \{c_1, \ \dots, \ c_c\}$ is a set of c application components and $R = \{r_1, \ \dots, \ r_r\}$ is a set of r RACs in SA

- I is the set of interactions between objects such that $I \subseteq \{(u, v, it, dt) | \ u, v \in O, \ it \in \{SYN, ASYN\}, dt \in \{SD, MD\}\}$. Here, it identifies the interaction type between objects u and v which can be either SYN for synchronous message communication or $ASYN$ for asynchronous message communication. Additionally, dt identifies the destination type for messages which can be either MD for a multicast message that is intended to multiple destinations or SD for a unicast message that is intended to a single destination.

9.1.1 Configuration Map

A configuration map defines the mapping between every object $o \in O$ in a software architecture SA to the node that is currently hosting this object (and vice versa). Formally, the configuration map M_{SA}^t of an architecture SA at time t is defined as $M_{SA}^t = \{N, O, \ H_n(k), H_c(k)\}$ where:

- $N = \{n_1, \dots, n_m\}$ is a set of m nodes in the software system
- $O = SA.O$ is the set of objects in the software system
- H_n(k) : N → O is a function that maps a node k ∈ N to the set $O_k \subseteq O$ of objects hosted by node k
- H_c(k): O → N is a function that maps an object $k \in O$ to the node $n \in N$ that is hosting object k

9.1.2 Architectural Patterns and Message-Based Transactions

In this dissertation, an architectural pattern AP in a software architecture SA is defined as $AP = \{O, I, T\}$ where:

- $O \subseteq SA.O$ is the set of objects in AP
- $I \subseteq SA.I$ is the set of interactions between the set of objects $AP.O$
- $T = \{t_1, \dots, t_t\}$ is a set of t transactions such that a transaction $t \in T$ is defined as $t = \{(e_a \rightarrow e_b)_1, \ \dots, \ (e_w \rightarrow e_z)_m\}$ of m dependencies between events executed by components $a, b, w, z \in AP.O$ where dependencies are defined based on Lamport's *happened-before* relationship (Lamport, 1978).

In this research, an event e can be one of the following message passing events:

- $send(msg_i^t, s, r)$ denotes an event executed by component s to send a message i of a transaction t to component r.
- $receive(msg_i^t, r, s)$ denotes the corresponding receive event executed by component r to receive msg_i^t that was sent by component s.

Additionally, an event e can be one of the following three events for manipulating the queues maintained by a RAC:

- $queue(msg_i^t, RAC_s, q)$ denotes an event executed by RAC_s to queue the message msg_i^t in a queue q.
- $move(msg_i^t, RAC_s, q1, q2)$ denotes an event executed by RAC_s to move the message msg_i^t from a queue $q1$ to queue $q2$.
- $dequeue(msg_i^t, RAC_s, q)$ denotes an event executed by RAC_s to dequeue the message msg_i^t from a queue q.

9.2 DARE Properties

This section describes several properties that the DARE framework ensures during failure recovery and dynamic adaptation. In particular, DARE ensures that when there is a failure, the system eventually recovers a configuration in which every failed component is recovered to a healthy node. For dynamic adaptation, DARE ensures that components are not adapted unless a region of quiescence has been established.

9.2.1 Failure Recovery

Let $n.s$ where $n \in M_{SA}^{t}.N$ denotes the current state of node n. Initially, $n.s = normal$ indicating that n is running normally. When a failure event occurs at n, then $n.s = failed$ indicating that n has failed.

The goal is to show that when a failure occurs to node $n \in M_{SA}^{t}.N$, then the software system eventually transitions from the configuration M_{SA}^{t} to configuration M_{SA}^{t+1} which has the following three properties:

- A1: $n.s = failed \Rightarrow \forall c \in M_{SA}^{t}.H_n(n)\big(\lozenge\, c \in M_{SA}^{t+1}.H_n(n')\big)\ s.t.\ n' \in M_{SA}^{t+1}.N\ and\ n' \neq n\ and\ n'.s = normal$
- A2: $n.s = failed \Rightarrow \lozenge\, \mathrm{n} \notin M_{SA}^{t+1}.N$
- A3: $n.s = failed \Rightarrow \forall \mathrm{n}_1, \mathrm{n}_2 \in M_{SA}^{t+1}.N\ s.t.\ \mathrm{n}_1 \neq \mathrm{n}_2,\ H_n(\mathrm{n}_1) \cap H_n(\mathrm{n}_2) = \emptyset$

The first property indicates that for every component c that was previously hosted by the failed node n according to the configuration map M_{SA}^{t}, eventually c is recovered to another node n' such that $n' \neq n$ and n' is in the $normal$ state. Property A2 indicates that the failed node n is not part of the new configuration map M_{SA}^{t+1}. Property A3 ensures a consistent configuration in which there are no duplicate application components in the configuration (i.e. a component cannot be hosted by more than one node). We assume that these three properties are satisfied during the initial deployment of the software systems. Thus, we show how our approach satisfies these properties at run-time in spite of failures.

We now show how these properties are satisfied.

Property A4: *By assumption if a node n has failed, then every healthy node eventually receives a notification message that node n has failed. Formally,*
$\forall n,\ n' \in M_{SA}^{t}.N\ s.t.\ n'.s = normal$, *the following property holds:*

- $\square(n.s = failed \Rightarrow \lozenge\ receive(suspected_n,\ FAM_{n'},\ DeSARM_{n'}\ ))$

This property indicates that when a failure event occurs at node n, then the Failure Analysis Manager (FAM) hosted by node n' whose current state is normal will eventually receive a notification message indicating failure of node n is suspected.

This property is provided by assumption, since it is assumed that DeSARM on every healthy handles sending notification messages to the peer FAM (located on the same node) when it suspects node failures.

Property A5: *By design, if the FAM on multiple nodes receive a notification message from DeSARM that the failure of a node is suspected, then only the node with the lowest IP address proceeds with recovery. Let* $activate(FAM_i, RAM_i, n)$ *be an event denoting FAM at node* i *activating RAM at node* i *to recover node* n*. Property A5 is defined based on property A4 as follows:*

- A5: $\Box\big(receive(suspected_n,\ FAM_{n'},\ DeSARM_{n'}\ ) \wedge \min(M_{SA}^{t}.N) = n' \Rightarrow \Diamond$ $activate(FAM_{n'}, RAM_{n'}, n)\big)$

<u>Proof.</u> Property A5 indicates that a FAM on node n' always eventually activates the RAM on node n' to recover the failed node n *only* if (1) FAM on node n' has received a notification message from DeSARM of failure of node n according to property A4 and node n' has the lowest identifier according to configuration M_{SA}^{t} (in this dissertation nodes are identified through their IP addresses). This property ensures that only one FAM is activated at any one moment to recover a failed node n and is achieved by design, since the state machine executed by the FAM (see fig. 5.2 in chapter 5):

- First retrieves the IP addresses of all nodes from the configuration manager
- Checks whether it is hosted by the node with the lowest IP address, and
- Activates the peer RAM if no heartbeat message is received from the suspected node.

<u>Proof of properties A1-A3</u>. From property A5, we can show that properties A1-A3 are satisfied as follows. By design, the RAM at node n' that satisfies property A5 proceeds with the recovery process by determining a recovery plan P to recover each failed component deployed to the failed node n according to the algorithm shown in fig 9.1. First, the RAM retrieves the software architecture SA from DeSARM and determines the failed components hosted by the failed node by looking up $M_{SA}^{t}.H_n(Failed\_IP\_Address)$ using the configuration manager. For each failed component c, the RAM determines the

$plan(c)$ to recover c, including notifying the RAC of the failed component and selecting the node that will host the recovered c.

```
Input:  Failed_IP_Address the IP address of the failed node
Effect: determines the recovery plan for recovering a component hosted by the failed
node.
Definitions:
    failed_components: the set of components hosted by the failed node.
    plan(x): the recovery plan for recovering a failed component x. plan(x) consists of
        the tuple (i, r, n) where:
        i is the input RAC that forward input messages, including synchronous
          requests and asynchronous messages, to component x.
        r is the set of recipient RACs for component x that receive
          output messages from component x.
        n is the IP address of the node to which component x must be recovered.
architecture ← SA  /* retrieved from DeSARM */
failed_components ← $M_{SA}^{t}.H_n(Failed\_IP\_Address)$  /* retrieved from local CM */
foreach c ∈ failed_components do
    plan(c).i ← t.u s.t. t ∈ SA AND t.v = c
    send $failed\ (c)\ to\ plan(c).i$
    plan(c).r ← t.v s.t. t ∈ architecture AND t.u = c
    plan(c).n ← $n''\ s.t. n'' \in random(M_{SA}^{t}.N) \wedge n''.s = \text{normal}$
```

```
**end**
```

Figure 9.1 algorithm executed by the RAM to determine a recovery plan

After determining the recovery plan, the RAM on node n' requests the RAM on node n'' to create an instance of failed component c and then connects c with the RACs that interact with it (see fig 9.2). Thus, this sequence of events satisfies property A1. The RAM on node n' then updates the configuration map by removing the failed node from the configuration and adding the new node to the configuration, which satisfies property A2. From property A5, only a single RAM is activated to recover a failed node n. Thus, a failed component c cannot be recovered by more than one RAM which ensures that c cannot be recovered on multiple nodes. As a result, property A3 is satisfied.

```
Input:  Failed_IP_Address the IP address of the failed node
        plan(c) the recovery plan to recover the failed component c
Effect: recovers the failed component c according to plan(c) by instantiating component c
on node $plan(c).n$, connecting c with plan(c).i and plan(c).r, updating the configuration,
and finally reactivating plan(c).i.
/* Create component c */
send $create(c)$ to RAM at node $plan(c).n$
/* Connect c */
send $connect(c, plan(x).n)$ to $plan(c).i$
```

```
foreach $i \in plan(c).r$ do
        $connect(i, M_{SA}^{t}.H_{c}(i))$ to $c$
end
//Update configuration map
$M_{SA}^{t}.\mathrm{N} = M_{SA}^{t}.\mathrm{N} - \{Failed\_IP\_Address\}$
$M_{SA}^{t}.\mathrm{N} = M_{SA}^{t}.\mathrm{N} \cup \{plan(c).n\}$
/* Activate c and input RAC */
send activate() to c
send reactivate() to $plan(c).i$
```
Figure 9.2 algorithm executed by the RAM to recover a failed component

9.2.2 Dynamic Adaptation

The adaptation capability of DARE is based on the change management rules described by Kramer and Magee (Kramer and Magee, 1990). Let $adapt(k) = \{cmd_1,, \dots, cmd_n\}$ be a set of n adaptation commands to adapt a component k such that an adaptation command cmd can be either:

- $Add(c, n)$ for adding component c to the architecture at node n.
- $Remove(c)$ for removing component c from the architecture.
- $Connect(c1, c2)$ for connecting component $c1$ with component $c2$.
- $Disconnect(c1, c2)$ for disconnecting component $c1$ from component $c2$.

DARE ensures before adapting component k, the recovery and adaptation connector of component k, denoted as RAC_k, steers component k to the quiescent state in which k is not engaged in any transactions and will not receive any new transactions from other components according to the following property:

Property A6: *When the Recovery and Adaptation Manager (RAM) receives an adaptation command to adapt component k, the RAM instructs RAC_k to passivate component k according to the following property:*

$$\Box\big(receive(adapt(k), RAM) \Rightarrow \Diamond\ send(Passivate, RAM, RAC_k\ )\big)$$

<u>*Proof.*</u> This property is satisfied by the state machine executed by the RAM (see Figure 5.3) since when the RAM receives an adaptation command, it sends the Passivate command to RAC_k and transitions to the Establishing Region of Quiescence state until it receives a response from RAC_k that component k has become quiescent. Therefore, no adaptation takes place unless k is in the quiescent state.

Once k has become quiescent, the RAM then proceeds with modifying architecture according to the following RAM commands:

1. $\forall add(c,\ n) \in adapt(k)$, DARE creates component c on node n
2. $\forall disconnect(c1,\ c2) \in adapt(k)$, DARE disconnects component $c1$ from $c2$
3. $\forall remove(c) \in adapt(k)$, DARE removes component c from the architecture
4. $\forall connect(c1, c2) \in adapt(k)$, DARE connects $c1$ to component $c2$

When all adaptation commands are performed, DARE sends the Activate command to every component that has been added in step 1 and then sends the Reactive command to RAC_k so that it resumes normal execution.

9.3 Recovery and Adaptation Properties of RAC for Stateless Components

This section describes several properties achieved by the Recovery and Adaptation Connector (RAC) for recovering and adapting *stateless* services during normal execution, failure recovery, and dynamic adaptation. The goal of a RAC_s that handles recovery and adaptation concerns of a stateless service s is to ensure the following property:

Property P1: *if RAC_s receives a client request, then eventually RAC_s sends the corresponding response of this request to the requesting client. Formally:*

$$\square\left(receive\left(request_i^t, RAC_s, Client_j\right) \Rightarrow \lozenge\, send\left(response_i^t, RAC_s, Client_j\right)\right)$$

9.3.1 Normal Execution

This section defines several properties of RAC_s during normal operation (i.e. assuming that there are no failures or adaptation).

Property P2: *any request received by RAC_s from a client is queued by RAC_s into the Service Pending Queue (SPQ). Formally:*

$$\square\left(receive\left(request_i^t, RAC_s, Client_j\right) \rightarrow queue(request_i^t, RAC_s, SPQ)\right)$$

Proof. This property is satisfied by design of the state machine executed by the Request Coordinator of RAC_s since the *first* action executed by the Request Coordinator of RAC_s whenever it receives a request from a client is to queue that request in the SPQ (c.f. fig. 3.4 in chapter 3). Formally, $receive\left(request_i^t,\ RequestCoodinator_s,\ Client_j\right) \rightarrow queue(request_i^t,\ RequestCoordinator_s,\ SPQ\ \ )$.

Property P3: *if RAC_s queued $request_i^t$ into the SPQ, then RAC_s will eventually forward this request to the service s. Formally:*

$$\Box\Big(queue(request_i^t,\ RAC_s,\ SPQ)\ \Rightarrow \Diamond\ send(request_i^t, RAC_s,\ s)\Big)$$

<u>*Proof.*</u> By design, when the Request Coordinator of RAC_s receives $request_i^t$, then in addition to queueing the request into the SPQ (as defined in property P2), the action is to also forward this request to Connector Control of RAC_s (c.f. fig. 3.4 in chapter 3). Formally, $receive\big(request_i^t,\ RequestCoordinator_s,\ Client_j\big) \rightarrow send(request_i^t,$ $RequestCoordinator_s,\ ConnectorControl_s)$. From the state machine executed by the Connector Control (c.f. fig. 3.3 in chapter 3), when Connector Control receives $request_i^t$ then the actions are to (1) forward the request to s and (2) instruct the Request Coordinator to move this request from the SPQ to the SAQ, i.e., $receive(request_i^t,$ $ConnectorControl_s,\ RequestCoordinator_s) \rightarrow$ $send(request_i^t,\ ConnectorControl_s,$ $s\ \ ) \wedge move(request_i^t,\ RequestCoordinator_s,\ SPQ, SAQ)$.

Property P4: *when RAC_s forwards $request_i^t$ to service s, then RAC_s moves this request from the SPQ to the Service Active Queue (SAQ), indicating that this request is currently being processed by s. Formally:*

$$\Box\Big(send(request_i^t,\ RAC_s,\ s)\ \rightarrow move(request_i^t,\ RAC_s,\ SPQ, SAQ)\Big)$$

<u>*Proof.*</u> By design:

$receive(request_i^t,\ ConnectorControl_s,\ RequestCoordinator_s) \rightarrow send(request_i^t,$

$ConnectorControl_s,$

$s\ \ ) \land send(request_i^t,\ ConnectorControl_s,\ RequestCoordinator_s)$

That is, receiving a client request causes Connector Control to send the request to the service s (as shown in property P3) and also to send this request to the Request Coordinator. By design, sending the request back to the Request Coordinator causes the Request Coordinator to move the request from the SPQ to the SAQ. I.e.

$receive(request_i^t,\ RequestCoordinator_s,\ Connector_s) \rightarrow move(request_i^t,\ RAC_s,$

$SPQ, SAQ).$

Property P5: *when RAC_s receives $response_i^t$ from service s, then this response is queued in the Response Forwarding Queue (RFQ). Formally:*

$$\square\Big(receive(response_i^t, RAC_s, s)\ \rightarrow queue(response_i^t, RAC_s, RFQ)\Big)$$

<u>*Proof*</u>. From property P3, RAC_s eventually forwards the requests it receives to service s. Since by assumption the service s is running normally, then s eventually sends to RAC_s the responses of the requests it receives from this connector. By design in the message sequence executed by RAC_s (see fig. 3.1 in chapter 3), the *first* action executed the Response Coordinator when it receives a service response is to queue that response in the Response Forwarding Queue (RFQ). Formally,

$receive(response_i^t,\ ResponseCoodinator_s,\ \ s) \rightarrow$

$queue(response_i^t,\ ResponseCoordinator_s,\ RFQ\ \ \ )$ .Thus, this message sequence satisfies property P5.

Property P6: *when RAC_s queues $response_i^t$ into RFQ, then this response is eventually forwarded by RAC_s to the requesting client. Formally:*

$$\Box\left(queue(response_i^t, RAC_s,\ RFQ)\ \Rightarrow \Diamond\ send\left(response_i^t,\ RAC_s,\ Client_j\right)\right)$$

<u>*Proof.*</u> By design, when the Response Coordinator of RAC_s receives a response, it forwards this response to Connector Control. In other words, $receive(response_i^t,$ $ResponseCoordinator_s,\ s) \rightarrow$ $send(response_i^t,\ ResponseCoordinator_s,\ ConnectorControl_s)$. This sequence of events causes the following message sequence to execute (see fig. 3.3 in chapter 3):

$receive(response_i^t,\ ConnectorControl_s,\ ResponseCoordinator_s) \rightarrow$

$send\left(response_i^t, ConnectorControl_s, Client_j\right)\ \land$

$send(response_i^t, ConnectorControl_s, RequestCoordinator_s) \land$

$send(response_i^t, ConnectorControl_s, ResponseCoordinator_s)$

This sequence of events indicates that when Connector Control receives $response_i^t$, it triggers actions to forward this response to (1) $Client_j$, (2) to the Request Coordinator, and (3) to the Response Coordinator.

Properties P7: *when RAC_s forwards $response_i^t$ to the requesting client, then this response is moved from the RFQ to the RRQ and the corresponding request of this response is moved from the SAQ to the SRQ. Formally:*

$\square\Big(send\big(response_i^t, RAC_s,\ \ Client_j\big)$

$\rightarrow\ move(response_i^t, RAC_s, RFQ, RRQ) \wedge move(request_i^t, RAC_s, SAQ, SRQ)\Big)$

<u>*Proof.*</u> By design and from the proof of the previous property:

$receive(response_i^t,\ ConnectorControl_s,\ ResponseCoordinator_s) \rightarrow$

$send\big(response_i^t,\ ConnectorControl_s,$

$Client_j\big)\ \wedge\ send(response_i^t,\ ConnectorControl_s,$

$RequestCoordinator_s)\ \wedge\ send(response_i^t,\ ConnectorControl_s,$

$ResponseCoordinator_s)$

That is, receiving a service response causes Connector Control to send the response to the requesting client and also sending this response to the Request and Response Coordinators. Sending the response back to the Request Coordinator causes the Request Coordinator to move the request from the SAQ to the SRQ. I.e. $receive(response_i^t,\ RequestCoordinator_s,\ Connector_s) \rightarrow move(request_i^t,$ $RequestCoordinator_s,\ SAQ, SRQ)$. Similarly, sending the response to the Response Coordinator causes this coordinator to move the response from the RFQ to the RRQ according to the sequence: $receive(response_i^t,\ ResponseCoordinator_s,$ $ConnectorConnector_s) \rightarrow move(response_i^t,\ ResponseCoordinator_s,\ RFQ, RRQ)$.

Properties P9: *RAC_s does not remove any transaction messages from the SRQ or the RRQ unless it receives an acknowledgement message from the client that initiated this transaction, indicating completion of the transaction. Formally:*

$$\square\Big(\big(\sim dequeue(request_i^t,\ RAC_s, SRQ)\ \wedge \quad \sim dequeue(response_i^t,\ RAC_s, RFQ)\big) \cup receive(ACK^t,\ RAC_s,\ Client_j)\Big)$$

<u>*Proof.*</u> This property is satisfied by the message sequence of the RAC design shown in fig. 3.1 in chapter 3, since the actions of dequeuing messages from these queues are dependent on receiving the ACK message from the client. In other words, $receive\big(ACK^t\ ,\ ConnectorControl_s,\ RequestCoordinator_s\big) \rightarrow send\big(transactionCompleted^t\ ,\ ConnectorControl_s,\ RequestCoordinator_s\big) \wedge send\big(transactionCompleted^t\ ,\ ConnectorControl_s,\ ResponseCoordinator_s\big)$. Receiving the $transactionCompleted^t$ message causes the Request and Response Coordinators to remove all messages of t from the SRQ and the RRQ.

9.3.2 Failure Recovery

This section defines several properties that are provided by RAC_s during failure recovery of service s. These properties ensure that any transaction that failed due to failure of s is eventually recovered and restarted when s is recovered. Furthermore, these properties ensure that all transactions received by the RAC while s is in the failed state are queued until s recovers from failure.

Properties P10: *Let $fail(s)$ indicates a failure event occurring to service s. when service s fails, then eventually RAC_s is notified of s failure by the Recovery and Adaptation Manager. Formally:*

$$\square\big(fail(s) \Rightarrow \lozenge\, receive(Failed_s, RAC_s,\ RAM_i)\big)$$

<u>*Proof*</u>. This property is satisfied by design as shown previously in section 9.2 since while determining the recovery plan, RAM_i always notifies the input RAC of the failed component of failure.

Property P11: *when RAC_s is notified by RAM_i of s failure, then RAC_s ceases forwarding messages to s and holds all requests to s in the SPQ. Formally:*

$$\square\big(RequestCoordinator_s.state = failed\ \wedge receive\big(request_i^t, RAC_s,\ Client_j\big) \rightarrow queue(request_i^t, RequestCoordinator_s, SPQ)\ \wedge \sim send(request_i^t, RequestCoordinator_s, ConnectorControl_s)\big)$$

<u>*Proof.*</u> This property is satisfied by the design of the Request Coordinator as can be clearly seen from the state machine executed by this coordinator (c.f. fig. 3.4 in chapter 3). In this state machine, when the Request Coordinator is notified of s failure according to property P10, then the Request Coordinator transitions to the Failed state in which the action for receiving client requests is to hold these requests into the SPQ. Formally,

$receive\left(Failed_s\ ,\ RequestCoordinator_s,\ ConnectoControl_s\right) \rightarrow$

$RequestCoordinator_s.state = Failed$

While the Request Coordinator is in the Failed state, then it holds all requests in the SPQ as follows:

$RequestCoordinator_s.state = Failed \wedge receive\left(request_i^t, RequestCoordinator_s,\right.$

$\left.Client_j\right) \rightarrow$

$queue(request_i^t, RequestCoordinator_s, SPQ)\ \wedge \sim$

$send(request_i^t, RequestCoordinator_s, ConnectorControl_s)$

Property P12: *Let $tFail(t)$ denote a failure event occurring to transaction t. RAC_s is capable of identifying a transaction t as failed according to the following property:*

$$\Box\left(tFail(t) \Rightarrow send(request_1^t,\ RAC_s,\ s)\ \wedge \sim receive\left(response_{final}^t,\ RAC_s,\ s\right)\right)$$

<u>*Proof.*</u> This property indicates that a transaction t is considered failed if RAC_s forwarded the *first* request of t to s but did not receive the *final* response of this transaction from s (due to s failure). We now show how RAC_s is capable of determining failed transactions.

From property P2 in section 9.3.1, a $request_i^t$ is first queued in the SPQ until it is forwarded to s. From property P4, a forwarded request is moved from the SPQ to the SAQ. From property P8 a forwarded request is moved from the SAQ to the SRQ when the corresponding response of this request is forwarded to the client. From property P9,

RAC_s does not remove any requests from SRQ until the transaction containing this request is completed.

Similarly, from property P5, RAC_s queues service responses in the RFQ until RAC_s forwards these responses to the requesting client. From property P7, RAC_s moves the responses it forwards from the RFQ to the RRQ. From property P9, RAC_s does not remove any responses from the SRQ until the transaction containing this response is completed.

Therefore, all requests and responses of active transactions are maintained by RAC_s in its queues. Given these properties, then RAC_s is able to determine failed transactions by analyzing transactions maintained in the SAQ and SRQ and determining whether the final response of each transaction is queued in either the RFQ or RRQ according to the algorithm shown in fig. 9.3. In this algorithm, RAC_s iterates over transactions queued in the SAQ and the SRQ. For every transaction t, if the final response of this transaction is *not* queued in either the RFQ or RRQ, then t is considered failed, which satisfies property P12.

```
failed ← { }
for each $\text{request}_i^t \in \text{SAQ} \cup \text{SRQ}$
    If $\text{response}_{final}^t \notin \text{RFQ} \cup \text{RRQ}$ then
        failed ← failed ∪ { t }
    end
```

```
end
```

Figure 9.3 algorithm executed by the RAC to determine failed transactions

Properties P13 RAC_s *is capable of recovering all identified failed transactions according to the property:*

- $\forall request_i^t \; s.t. t \in failed, moveToHead(\text{request}_i^t, RAC_s, \; SAQ \cup SRQ \,, SPQ)$

__Proof__. Property P13 indicates that RAC_s moves the requests of every failed transaction from the SRQ and the SAQ to the head of the SPQ.

To ensure this property, RAC_s recovers failed transactions according to the algorithm shown in fig 9.4. First, RAC_s iterate over the requests queued in the SAQ. For each request of a failed transaction in the SAQ, RAC_s moves the request from the SAQ to the head of the SPQ. Then RAC_s iterate over the requests queued in the SRQ. For each request in the SRQ of a failed transaction, RAC_s moves the request from the SRQ to the head of the SPQ.

```
for each $\mathrm{request}_i^t \in \mathrm{SAQ}$
    If t $\in$ failed then
        moveToHead($\mathrm{request}_i^t, SAQ, SPQ$)
    end
end
for each $\mathrm{request}_i^t \in \mathrm{SRQ}$
    If t $\in$ failed then
        moveToHead($\mathrm{request}_i^t, SRQ, SPQ$)
    end
end
```

Figure 9.4 algorithm executed by the RAC to recover failed transactions

Properties P15: *let $recovered(s, n)$ indicate recovery of component s to node n. When component s is recovered, RAC_s is eventually reactivated. Formally:*

$$\square(recovered(s, n) \Rightarrow \lozenge\, receive(reactivate, RAC_s,\ RAM_i))$$

<u>*Proof*</u>. This property is satisfied by design since while executing the recovery plan, the Recovery and Adaptation Manager RAM_i always notifies the input RAC of the recovered component of component recovery as shown previously in section 9.2 (see fig. 9.2).

Properties P16: *when RAC_s is reactivated after s is recovered, then RAC_s resumes sending messages, including held and lost messages, to s. Formally, $\forall request_i^t \in SPQ$:*

$$\square(receive(reactivate, RAC_s, RAM_i) \Rightarrow \Diamond\, send(request_i^t, RequestCoordinator_s,\quad ConnectorControl_s))$$

This property indicates that when RAC_s is reactivated, then eventually $RequestCoordinator_s$ forwards all requests queued in its SPQ to $ConnectorControl_s$, including all requests held by $RequestCoordinator_s$ (according to property P10) and recovered requests (according to property P13). This property is ensured by design as shown in the state machine executed by the Request Coordinator (see fig. 3.4 in chapter 3) since when this coordinator is reactivated, it sends all messages queued in the SPQ to the Connector Control. Connector Control then forwards these requests to s normally according to property P3.

9.3.3 Dynamic Adaptation

This section defines several properties that are provided by RAC_s during adaptation of service s. These properties ensure that s can be adapted only if s has completed all transactions that it is currently engaged and will not receive any new transactions from other components. Furthermore, these properties ensure that any new transactions received by the RAC while the component is being adapted are queued until dynamic adaptation is completed.

Properties P17, P18: *when RAC_s receives the Passivate command from RAM_i, then RAC_s eventually transitions to the Quiescent state. Furthermore, RAC_s does not transition to the Quiescent state unless the service is not engaged in any transactions:*

$$\square(receive(passivate, RAC_s,\ RAM_i) \Rightarrow \Diamond\, RAC_s.state = quiescent)$$

$$\square\left((RAC_s.state \cong \text{Quiescent})\ \cup RAC_s.\text{ActiveTransactionsCount} = 0)\right)$$

<u>Proof.</u> From the design of the state machine executed by Connector Control of RAC_s (see figure 3.3 in chapter 3), Connector Control maintains the number of active transactions that s is currently engaged according to the following rules:

- $send(reqeust_1^t, RAC_s, s) \rightarrow increment(ActiveTransactionCount)$
- $receive\left(response_{final}^t, RAC_s, s\right) \rightarrow$
  $decrement(ActiveTransactionCount)$

By design, if Connector Control of RAC_s receives the Passivate command while $ActiveTransactionCount = 0$, then Connector Control transitions immediately to the $quiescent$ state (see fig. 3.3 in chapter 3). Otherwise, Connector Control transitions to the intermediate $Passivating$ state in which it allows existing transactions to terminate normally but does not forward any new transactions to s according to the following property:

- P19: $\square\ (RequestCoordinator_s.state =$
  $Passivating\ \wedge receive\left(request_1^t, RAC_s,\ Client_j\right) \rightarrow$
  $queue(request_1^t, RequestCoordinator_s, SPQ)\ \wedge \sim$
  $send(request_1^t, RequestCoordinator_s, ConnectorControl_s))$

This property is ensured by the state machine executed by $RequestCoordinator_s$ since the action for receiving requests that initiate new transaction with s is to hold the request in the SPQ. Thus, this property ensures that eventually $ActiveTransactionCount = 0$ and that RAC_s transitions from the $Passivating$ to the $Quiescent$ state.

Properties P20: *while in the Quiescent state, RAC_s ceases forwarding all requests it receives from clients to s and holds these requests in the SPQ. Formally:*

$$\Box\left(receive\left(reqeust_1^t, RAC_s, Client_j\right) \wedge RAC_s.state = Quiescent \rightarrow queue(request_i^t, RequestCoordinator_s, SPQ) \wedge \sim send(reqeust_1^t, RAC_s, s)\right)$$

Proof. This property is satisfied by the design of the Request Coordinator as can be clearly seen from the state machine executed by this coordinator (c.f. fig. 3.4 in chapter 3). In this state machine, when the Request Coordinator transitions eventually to the Quiescent state according to property P18, then the action while in the Quiescent state for receiving client requests is to hold these requests into the SPQ. Formally,

$receive\left(NotifyQuiescent_s\ ,\ RequestCoordinator_s,\ ConnectoControl_s\right) \rightarrow RequestCoordinator_s.state = Quiescent$

When the Request Coordinator receives a request while it is in the Quiescent state, the action is to hold that request in the SPQ (c.f. fig. 3.4 in chapter 3)

Properties P21: *let $adapted(s)$ denotes completion of service s adaptation. When s adaptation is completed, then RAC_s is eventually reactivated. Formally:*

$$\Box(adapted(s) \Rightarrow \Diamond\, receive(reactivate, RAC_s,\ RAM_i))$$

Proof. This property is satisfied by design since when the Recovery and Adaptation Manager RAM_i finishes adapting s, the RAM always reactive the input RAC of the component affected by adaptation (see section 9.2).

Properties P22: *when RAC_s is reactivated after s is adapted to s', then RAC_s resumes sending messages, including held messages, to s'. Formally,* $\forall request_i^t \in SPQ$:

$$\square(receive(reactivate, RAC_s, RAM_i) \Rightarrow \lozenge\, send(request_i^t, RequestCoordinator_s, \quad ConnectorControl_s))$$

This property indicates that when RAC_s is reactivated, then eventually $RequestCoordinator_s$ forwards all requests queued in its SPQ to $ConnectorControl_s$, including all requests held by $RequestCoordinator_s$ (according to properties 19 and 20). This property is ensured by design as shown in the state machine executed by the Request Coordinator (see fig. 3.4 in chapter 3) since when this coordinator is reactivated, it sends all messages queued in the SPQ to the Connector Control. Connector Control then forwards these requests to s' normally according to property P3.

9.4 Recovery and Adaptation Properties of RAC for Stateful Components

This section discusses several properties ensured by the RAC for handling adaptation and recovery concerns of *stateful* components in SOA patterns. In this dissertation, the state of a stateful component c at time k is viewed as the set $S_c^k = \{t_1, \dots, t_n\}$ of n transactions executed by c. Furthermore, every transaction $t \in S_c^k$ has a status $t.s$ such that $t.s \in \{Active, Committed, Aborted, Prepared\}$ where:

- $Active$ indicates that t is currently being executed by c and has not yet been completed.
- $Committed$ indicates that t has executed to its entirety.

- $Aborted$ indicates that t has been aborted and must be rolled back, as defined below.
- $Prepared$ indicates that *t* has committed *provisionally*. A prepared transaction remains in the $Prepared$ status until the RAC of component c sends either the *Commit* or *Abort* message to c so that c either commits or aborts t, respectively.

We assume that the state of the stateful component c is maintained by a transactional processing system TP that ensures atomicity of every transaction executed by c. Formally, TP ensures during normal execution that c always transitions from a state S_c^k to state S_c^{k+1} with the following properties:

- AS1: $\forall \mathrm{t} \in S_c^k,$ s.t. $t.s = Committed \vee Prepared \vee \mathrm{Active} \Rightarrow t \in S_c^{k+1}$
- AS2: $\forall \mathrm{t} \in S_c^k,$ s.t. $t.s = Aborted \Rightarrow t \notin S_c^{t+1}$

Assumption AS1 indicates that TP ensures the durability of committed transactions. Furthermore, AS1 ensures that prepared and active transactions are maintained until they are either committed or aborted. Assumption AS2 ensures that aborted transactions are rolled back from the state of component c.

Furthermore, we assume that when c fails at time t^f and then subsequently recovers at time t^r (as defined in section 9.2), TP recovers the state, denoted S_c^r as follows:

- AS3: $\forall \mathrm{t} \in S_c^f$ s.t. $t.s = Committed \vee Prepared \Rightarrow t \in S_c^r$
- AS4: $\forall \mathrm{t} \in S_c^f$ s.t. $t.s = Aborted \vee Active \Rightarrow t \notin S_c^r$

These assumptions indicate that TP is capable of reconstructing the state of c by recovering committed and prepared transactions (AS3) and rolling back transactions that were in the aborted or active statuses when the failure occurred (AS4).

Given these assumptions of TP, the goal of RAC_c that handles adaptation and recovery concerns of the stateful component c is to ensure the following properties:

Property S1: if RAC_c receives a client request, then eventually RAC_c sends the corresponding response of this request to the requesting client. *Formally:*

$$\square\left(receive\left(request_i^t, RAC_c,\ Client_j\right) \Rightarrow \lozenge\ send\left(response_i^t, RAC_c,\ Client_j\right)\right)$$

This property is similar to property P1 in section 9.3.1. However, in addition to this property, RAC_c must also ensure the following property:

Property S2: eventually, all transactions that were rolled back by TP due to c failure are restarted by RAC_c. *Formally,* $\forall \mathrm{t} \in S_c^f$*:*

$$\square\big((t.s = Active\ \vee Prepared) \Rightarrow \lozenge\ (t.s = Committed\ \vee Aborted)\big)$$

The goal is to ensure these properties during normal execution, failure recovery, and adaptation, as will be explained in the next subsections.

9.4.1 Normal Execution

The Two-Phase Protocol (Bernstein and Newcomer, 2009) defines the following control message:

- $PrepareToCommit^t$ denotes a request for the stateful component to prepare to commit the transaction t. We assume that $PrepareToCommit^t$ piggybacks a request, denoted as $PrepareToCommit^t.request$, which contains the update operation to be performed by c.

- $ReadyToCommit^t$ denotes the corresponding response of the $PrepareToCommit^t$ request indicating that the stateful component has prepared to commit transaction t. We assume that the $ReadyToCommit^t$ message piggybacks a response, denoted as $ReadyToCommit^t.response$, which contains the result of performing the update operation.
- $RefuseToCommit^t$ denotes an alternative response to the $PrepareToCommit^t$ request indicating that the stateful component is unable to prepare to commit transaction *t*.
- $Commit^t$ denotes a request for the stateful component to commit the transaction t.
- $Committed^t$ denotes a response for a $Commit^t$ request that transaction t is committed.
- $Abort^t$ denotes a request for the stateful component to abort the transaction t.
- $Aborted^t$ denotes a response for an $Abort^t$request that transaction t is aborted.

Formally, these control messages manipulate the status of a transaction t ∈ S_c^t as follows:

- AS5: $\Box\Big(receive\big(prepareToCommit^{t}\ ,\ \ c,\ \ RAC_{c}\big)\ \wedge t.s = Active \rightarrow \Big(\big(t.s = Prepared\ \ \wedge send(readyToCommit^{t}\ , c, RAC_{c}\big) \vee\ \big(t.s = Aborted \wedge\ send(refuseToCommit^{t}\ , c, RAC_{c})\big)\Big)\Big)$
- AS6: $\Box\Big(receive\big(commit^{t}\ ,\ \ c,\ \ RAC_{c}\big)\ \wedge t.s = Prepared \rightarrow \Big(t.s = committed \wedge send\big(committed^{t}\ ,\ c,\ \ RAC_{c}\big)\Big)\Big)$
- AS7: $\Box\Big(receive\big(abort^{t}\ ,\ \ c,\ \ RAC_{c}\big)\ \wedge t.s = (Prepared\ \ \vee Aborted) \rightarrow \Big(t.s = aborted \wedge send\big(aborted^{t}\ ,\ c,\ \ RAC_{c}\big)\Big)\Big)$

Assumption AS5 indicates that if component c receives a $Prepare\ To\ Commit$ message for a transaction t that is in the $Active$ state, then eventually either (1) the status of t at component c becomes $Prepared$ and c sends back the $Ready\ To\ Commit$ message or (2) the status of t at component c becomes $Aborted$ and c sends the $Refuse\ To\ Commit$ message. AS6 defines behavior when c receives a $Commit$ request for a transaction t that is in the $Prepared$ status in which the status of this transaction becomes $Committed$ and c sends the $Commited$ message. AS7 defines behavior when c receives a $Abort$ request for a transaction t that is in the $Prepared$ status in which the status of this transaction becomes $Aborted$ and c sends the $Aborted$ message.

Property S3: *any request received by RAC_c from a client is queued by RAC_c into the Service Pending Queue (SPQ). Formally:*

$$\Box\left(receive\left(request_i^t, RAC_c, Client_j\right) \rightarrow queue\left(request_i^t, RAC_c, SPQ\right)\right)$$

Proof. This property is satisfied by design of the state machine executed by the Request Coordinator of RAC_c since the *first* action executed by the Request Coordinator whenever it receives a request from a client is to queue that request in the SPQ (c.f. fig. 3.4 in chapter 3). Formally, $receive\left(request_i^t,\ RequestCoodinator_c,\ Client_j\right) \rightarrow queue\left(request_i^t,\ RequestCoordinator_c,\ SPQ\ \ \right)$.

Property S4: *if RAC_c queued $request^t$ into the SPQ, then RAC_c will eventually request c to prepare to commit t. Formally:*

$$\Box\left(queue\left(request^t\ ,\ RAC_c,\ SPQ\right)\ \Rightarrow \lozenge\ send\left(prepareToCommit^t\ , RAC_c,\ c\right)\right)$$

Proof. By design, when the Request Coordinator of RAC_c receives $request^t$, then in addition to queueing the request into the SPQ, the action is to also forward this request to Connector Control of RAC_c (c.f. fig. 3.4 in chapter 3). Formally, $receive\left(request^t\ ,\ RequestCoordinator_c,\ Client_j\right) \rightarrow send\left(request^t\ ,\ RequestCoordinator_c,\ ConnectorControl_c\right)$. From the state machine executed by the Connector Control (c.f. fig. 3.6 in chapter 3), when Connector Control receives $request^t$ then the actions are to (1) instruct c to prepare to commit this transaction and (2) instruct the Request Coordinator to also prepare to commit this transaction, i.e., $receive\left(request^t\ ,\ ConnectorControl_c,\ RequestCoordinator_c\right) \rightarrow send\left(PrepareToCommit^t\ ,\right.$

$ConnectorControl_c,\ c\ \ ) \wedge send(PrepareToCommit^t\ ,\ ConnectorControl_c,$

$RequestCoordinator_c)$

Property S5: *For each transaction that is being prepared to commit at c,* RAC_c *queues the Prepare To Commit message of the transaction into the SAQ. Formally:*

$$\Box(send(prepareTCommit^t\ ,\ RAC_c,\ c) \rightarrow dequeue(request^t\ ,\ RAC_c,$$

$$SPQ)\ \wedge queue(prepareToCommit^t\ ,\ RAC_c,\ SAQ))$$

<u>*Proof.*</u> By design and as shown in the previous property:

$receive(request^t\ ,\ ConnectorControl_c,\ RequestCoordinator_c) \rightarrow$

$send(prepareToCommit^t\ ,\ ConnectorControl_c,$

$c\ \ )\ \wedge\ send(prepareToCommit^t\ ,\ ConnectorControl_c,\ RequestCoordinator_c)$

That is, receiving a client request causes Connector Control to send the Prepare To Commit message to the c and also to send this message to the Request Coordinator. By design, sending the Prepare To Commit message to the Request Coordinator causes the Request Coordinator to dequeue the request of this transaction from the SPQ and queue the Prepare To Commit message into the SAQ. I.e.

$receive(PrepareToCommit^t\ ,\ RequestCoordinator_c,\ ConnectorControl_c) \rightarrow$

$dequeue(request^t\ ,\ RequestCoordinator_c,\ SPQ)\ \wedge queue(prepareToCommit^t\ ,$

$RequestCoordinator_c,\ SAQ)$

Properties S6 and S7: *when RAC_c receives $ReadyToCommit^t$ or $RefuseToCommit^t$ from c, then this response is queued in the Response Forwarding Queue (RFQ). Formally:*

- $\square\left(receive\left(readyToCommit^t, RAC_c,\ c\right) \rightarrow queue\left(readyToCommit^t, RAC_c,\ RFQ\right)\right)$
- $\square\left(receive\left(refuseToCommit^t, RAC_c, c\right) \rightarrow queue\left(refuseToCommit^t, RAC_c, RFQ\right)\right)$

<u>*Proof*</u>. From property S4, RAC_c eventually forwards a Prepare To Commit message to c. By assumption AS5, c eventually sends either the Prepare To Commit or Refuse To Commit message back to RAC_c. By design in the message sequence executed by RAC_c (see fig. 3.1 in chapter 3), the *first* action executed the Response Coordinator when it receives a service response is to queue that response in the Response Forwarding Queue (RFQ). Formally,

$receive\left(response^t,\ ResponseCoodinator_c,\ c\ \right) \rightarrow queue\left(response^t,\ ResponseCoordinator_c,\ RFQ\ \right)$ .Thus, this message sequence satisfies property S6 and S7.

A transaction t involving a stateful component c can be either distributed or not distributed (see chapter 3). Let $control^{t}$ be any 2PC control message and $control^{t}.isDistributed$ be a flag indicating whether t is a distributed transaction.

Properties S8: *when RAC_c queues a Ready To Commit (or Refuse To Commit) into RFQ for a non-distributed transaction, then RAC_c eventually sends the Commit (or Abort) message to c. Formally:*

- $\square\Big(queue\big(readyToCommit^{t}\ , RAC_c,$ $RFQ\big)\wedge \sim readyToCommit^{t}\ .isDistributed\ \Rightarrow \lozenge\ send\big(commit^{t}\ ,$ $RAC_c,\ \ c\ \ \big)\Big)$
- $\square\Big(queue\big(refuseToCommit^{t}\ , RAC_c, RFQ\big)\wedge \sim$ $readyToCommit^{t}\ .isDistributed\ \Rightarrow \lozenge\ send\big(abort^{t}\ ,\ RAC_c,\ \ c\ \ \big)\Big)$

<u>*Proof.*</u> By design, when the Response Coordinator of RAC_c receives a Ready To Commit or Refuse To Commit response for a non-distributed transaction, it forwards this response to Connector Control. In other words,
$receive\big(readyToCommit^{t}\ ,\ ResponseCoordinator_c,\ c\big) \rightarrow$
$send\big(readyToCommit^{t}\ ,\ ResponseCoordinator_c,\ ConnectorControl_c\big)$ and

$receive(refuseToCommit^t\ ,\ ResponseCoordinator_c,\ c) \rightarrow$

$send(refuseToCommit^t\ ,\ ResponseCoordinator_c,\ ConnectorControl_c)$. This sequence causes the following message sequences to execute (see fig. 3.6 in chapter 3):

$receive(readyToCommit^t\ ,\ ConnectorControl_c,\ ResponseCoordinator_c) \wedge \sim$

$readyToCommit^t\ .isDistributed \rightarrow send(commit^t\ ,ConnectorControl_c, c\ \ )$

$\wedge\, send(commit^t\ ,ConnectorControl_c, RequestCoordinator_c)$ and

$receive(refuseToCommit^t\ ,\ ConnectorControl_c,\ ResponseCoordinator_c) \wedge \sim$

$readyToCommit^t\ .isDistributed \rightarrow send(abort^t\ ,ConnectorControl_c, c\ \ )$

$\wedge\, send(abort^t\ ,ConnectorControl_c, RequestCoordinator_c)$

Properties S9-S12: *when RAC_c queues a Ready To Commit or Refuse To Commit into RFQ for a distributed transaction, then RAC_c eventually sends this response to the coordinator of the distributed transaction. Additionally, when RAC_c receives the decision of the coordinator of the distributed transaction, it forwards this decision to c. Formally:*

- $\square\Big(queue(readyToCommit^t\ ,RAC_c,$

  $RFQ) \wedge readyToCommit^t\ .isDistributed\ \Rightarrow$

  $\lozenge\, send(readyToCommit^t\ ,RAC_c,\ Coordinator_j)\Big)$

-

  $\square\Big(queue(refuseToCommit^t\ ,RAC_c,$

$RFQ)\wedge readyToCommit^{t}.isDistributed \Rightarrow$

$\lozenge\, send\big(refuseToCommit^{t}, RAC_c,\ Coordinator_j)\big)$

- $\square\Big(receive\big(commit^{t}, RAC_c,\ Coordinator_j\big) \Rightarrow \lozenge\, send\big(commit^{t}, RAC_c,$ $c)\big)\Big)$
- $\square\Big(receive\big(abort^{t}, RAC_c,\ Coordinator_j\big) \Rightarrow \lozenge\, send\big(abort^{t}, RAC_c,\ c\big)\Big)$

<u>*Proof.*</u> By design, the state machine executed by Connector Control of RAC_c (see fig. 3.14 in chapter 3) forwards responses of distributed transactions to the coordinator of the distributed transaction and does not forward a decision to c unless it has received this decision from the coordinator of the distributed transaction. As a result, by the 2PC protocol, the coordinator of the distributed transaction eventually sends either the Commit or Abort decision to RAC_c. By design, when RAC_c receives this decision message from the coordinator, it forwards this decision to c. Formally, $receive\big(Commit^{t},\ ConnectorControl_c,\ RequestCoordinator_c\big) \rightarrow$ $send\big(Commit^{t},\ ConnectorControl_c,\ c\ \big)$ and $receive\big(Abort^{t},\ ConnectorControl_c,\ RequestCoordinator_c\big) \rightarrow$ $send\big(Abort^{t},\ ConnectorControl_c,\ c\ \big)$

Property S13-S14: *when RAC_c requests c to commit a transaction during the second phase of 2PC protocol, then RAC_c queues the Commit message into the SRQ. Similarly,* *when RAC_c requests c to abort a transaction during the second phase of 2PC protocol, then RAC_c queues the Abort message into the SRQ. Formally:*

- $\square\big(send\big(Commit^t\ ,\ RAC_c,\ c\big) \wedge readyToCommit^t\ \in RFQ\ \cup RRQ\ \rightarrow dequeue\big(prepareToCommit^t\ ,\ RAC_c,\ SAQ\big)\ \wedge queue\big(Commit^t\ ,\ RAC_c,\ SRQ\big)\big)$
- $\square\big(send\big(Abort^t\ ,\ RAC_c,\ c\big) \wedge (readyToCommit^t\ \vee\ refuseToCommit^t\ ) \in RFQ\ \cup RRQ\ \rightarrow dequeue\big(prepareToCommit^t\ ,\ RAC_c,\ SAQ\big)\ \wedge queue\big(Abort^t\ ,\ RAC_c,\ SRQ\big)\big)$

<u>*Proof.*</u> By design when Connector Control forwards the Commit or Abort message to c (as shown in properties S8-12), the action is to also request the Request Coordinator to either Commit or Abort this transaction. By design, receiving the Commit or Abort message from Connector Control causes the following sequence of events at the Request Coordinator:

- $receive\big(commit^t\ , RequestCoordinator_c\ \ , ConnectorControl_c\ \big) \rightarrow dequeue\big(prepareToCommit^t\ ,\ RequestCoordinator_c,\ SAQ\big)\ \wedge queue\big(commit^t\ ,\ RequestCoordinator_c,\ SRQ\big)$
- $receive\big(abort^t\ , RequestCoordinator_c\ \ , ConnectorControl_c\ \big) \rightarrow dequeue\big(prepareToCommit^t\ ,\ RequestCoordinator_c,\ SAQ\big)\ \wedge queue\big(abort^t\ ,\ RequestCoordinator_c,\ SRQ\big)$

Properties S15 and S16: *when RAC_c receives $Committed^t$ or $Aborted^t$ from c for a transaction that is in the second phase of 2PC protocol, then this response is queued in the Response Forwarding Queue (RFQ). Formally:*

$$\square\left(receive\left(committed^t, RAC_c, c\right) \wedge \quad readyToCommit^t \in RFQ \cup RRQ \rightarrow queue\left(committed^t, RAC_c, RFQ\right)\right)$$

$$\square\left(receive\left(aborted^t, RAC_c, c\right) \wedge \quad (readyToCommit^t \vee refuseToCommit^t) \in RFQ \cup RRQ \rightarrow queue\left(aborted^t, RAC_c, RFQ\right)\right)$$

Proof. From the previous properties, RAC_c eventually forwards a Commit or Abort message to c for each prepared transaction at c. By assumptions AS6 and AS7, c eventually sends either the Committed or Aborted response back to RAC_c. By design in the message sequence executed by RAC_c (see fig. 3.1 in chapter 3), the *first* action executed the Response Coordinator when it receives a response for a transaction that is currently in the second phase of 2PC is to queue that response in the Response Forwarding Queue (RFQ).

Properties S17: *for non-distributed transactions, RAC_s eventually sends the final response of transactions to the requesting client. Formally:*

- $\square\left(queue\left(committed^t, RAC_c, RFQ\right) \wedge \sim committed^t.isDistributed \Rightarrow \lozenge\, send\left(readyToCommit^t.response, RAC_c, Client_j\right)\right)$

- $\Box\Big(queue\big(aborted^t\ , RAC_c,\ RFQ\big) \wedge \sim aborted^t\ .isDistributed\ \Rightarrow \Diamond$ $send\big(refuseToCommit^t\ .response,\ RAC_c,\ \ Client_j\big)\Big)$

<u>*Proof.*</u> By design, when the Response Coordinator of RAC_c receives Committed or aborted response for a non-distributed transaction from c, it forwards this response to Connector Control. In other words, $receive\big(Committed^t\ ,\ ResponseCoordinator_c,$ $c\big) \to send\big(Committed^t\ ,\ ResponseCoordinator_c, ConnectorControl_c\big)$ and $receive\big(Aborted^t\ ,\ ResponseCoordinator_c,\ c\big) \to$ $send\big(Aborted^t\ ,\ ResponseCoordinator_c, ConnectorControl_c\big)$. These sequence of events causes the following message sequence to execute (see fig. 3.6 in chapter 3): $receive\big(Committed^t\ ,\ ConnectorControl_c,\ ResponseCoordinator_c\big) \to$ $send\big(PrapareToCommit^t\ .response, ConnectorControl_c, Client_j\big)$ and $receive\big(Aborted^t\ ,\ ConnectorControl_c,\ ResponseCoordinator_c\big) \to$ $send\big(RefuseToCommit^t\ .response, ConnectorControl_c, Client_j\big)$

Properties S18 and S19: *for distributed transactions, RAC_c eventually sends the Committed or Aborted response of committed or aborted transactions to the coordinator of the distributed transaction. Formally:*

$\Box\Big(queue\big(committed^t\ , RAC_c,\ RFQ\big) \wedge committed^t\ .isDistributed\ \Rightarrow \Diamond$ $send\big(committed^t\ ,\ RAC_c,\ \ Coordinator_j\big)\Big)$

$\Box\left(queue\left(aborted^{t}\ , RAC_{c},\ RFQ\right) \wedge aborted^{t}\ .isDistributed\ \Rightarrow \Diamond\ send\left(aborted^{t}\ ,\right.\right.$ $\left.\left. RAC_{c},\ \ Coordinator_{j}\right)\right)$

Proof. By design, when the Response Coordinator of RAC_c receives either a Committed or Aborted response, it forwards this response to Connector Control. In other words, $receive\left(Committed^{t}\ ,\ ResponseCoordinator_{c},\ c\right) \rightarrow$ $send\left(Committed^{t}\ ,\ \ ResponseCoordinator_{c},\ ConnectorControl_{c}\right)$ and $receive\left(Aborted^{t}\ ,\ ResponseCoordinator_{c},\ c\right) \rightarrow$ $send\left(Aborted^{t}\ ,\ \ ResponseCoordinator_{c},\ ConnectorControl_{c}\right)$. For a Committed or Aborted response such that this response is of a distributed transaction, this sequence of events causes the following message sequence to execute (see fig. 3.14 in chapter 3): $receive\left(Committed^{t}\ ,\ ConnectorControl_{c},\ ResponseCoordinator_{c}\right) \rightarrow$ $send\left(Committed^{t}\ , ConnectorControl_{c}, Coordinator_{j}\right)$ and $receive\left(Aborted^{t}\ ,\right.$ $\left. ConnectorControl_{c},\ ResponseCoordinator_{c}\right) \rightarrow$ $send\left(Aborted^{t}\ , ConnectorControl_{c}, Coordinator_{j}\right)$

These two sequence of events indicates that when Connector Control receives either $Committed^{t}$ or $Aborted^{t}$ response, it triggers actions to forward this response to the coordinator of the distributed transaction.

9.4.2 Failure Recovery

This section defines several properties that are provided by RAC_c during failure recovery of service c. These properties ensure that any transaction that failed due to failure of c is eventually recovered when c is recovered. Furthermore, these properties ensure that all transactions received by the RAC while c is in the failed state are queued until c recovers from failure.

Property S25: *Let $tFail(t)$ denotes a failure event occurring to transaction t. RAC_c is capable of identifying a transaction t as failed according to the following property:*

$$\Box\left(tFail(t) \Rightarrow send(request_1^t,\ RAC_c,\ c)\ \wedge \sim\left(receive\left(committed^t\ ,\ RAC_c,\ c\right) \vee receive\left(aborted^t\ ,\ RAC_c,\ c\right)\right)\right)$$

This property indicates that a transaction t is considered failed if RAC_c forwarded the *first* request of t to c and RAC_c did receive either the *committed* or *aborted* response of this transaction during phase 2 of the 2PC protocol from c (due to c failure). We now show how RAC_c is capable of determining all failed transactions. From property P10, RAC_c does not remove transaction messages from its queues unless it receives an ACK^t from the component that initiated t indicating that t has completed.

Given that RAC_c does not discard any message of a transaction t unless t has completed, then RAC_c is able to determine failed transactions, according to property S25 by analyzing transactions maintained in the SAQ and SRQ and determining whether a committed or aborted response for each transaction at the second phase of 2PC is queued in either the RFQ or RRQ (see fig. 9.5).

```
failed ← { }
for each $\mathrm{msg}_i^t \in \mathrm{SAQ} \cup \mathrm{SRQ}$
    if $\mathrm{ReadyToCommit}^t \vee \mathrm{RefuseToCommit}^t \notin \mathrm{RFQ} \cup \mathrm{RRQ}$ then
        failed ← failed ∪ { t }
    else if $\mathrm{Committed}^t \vee \mathrm{Aborted}^t \notin \mathrm{RFQ} \cup \mathrm{RRQ}$ then
        failed ← failed ∪ { t }
    end
end
```

Figure 9.5 algorithm executed by the RAC to determine failed 2PC transactions

Properties S26 and S27: *RAC_c is capable of recovering all failed 2PC transactions such that $\forall request^t \in (SAQ \cup SRQ)\ s.t.\ t \in failed$:*

- S26: $\square\Big(prepareToCommit^t \in SAQ \wedge \big(readyToCommit^t \vee refuseToCommit^t\big) \notin RFQ \cup RRQ \Rightarrow \lozenge\, send\big(abort^t, RAC_c, c\big)\Big)$
- S27: $\square\Big(commit^t \in SRQ \wedge committed^t \notin RFQ \cup RRQ \Rightarrow \lozenge\, send\big(commit^t, RAC_c, c\big)\Big)$
- S28:
  $\square\Big(abort^t \in SRQ \wedge aborted^t \notin RFQ \cup RRQ \Rightarrow \lozenge\, send\big(abort^t, RAC_c, c\big)\Big)$

In addition, distributed transactions require the following properties:

- S29:

  $\square\Big(readyToCommit^{t} \in$

  $RFQ \wedge readyToCommit^{t}\ .IsDistributed \wedge (Commit^{t} \ \vee\ Abort^{t}\ ) \notin$

  $SRQ \ \Rightarrow \lozenge\ send\big(readyToCommit^{t}\ ,\ RAC_{c},\ Coordinator_{j}\ \big)\Big)$

- S30:

  $\square\Big(refuseToCommit^{t} \in$

  $RFQ \wedge refuseToCommit^{t}\ .IsDistributed \wedge\ Abort^{t} \ \notin SRQ \ \Rightarrow \lozenge$

  $send\big(refuseToCommit^{t}\ ,\ RAC_{c},\ Coordinator_{j}\ \big)\Big)$

- S31: $\square\Big(committed^{t} \ \in RFQ \wedge commited^{t}\ .IsDistributed \Rightarrow \lozenge$

  $send\big(committed^{t}\ ,\ RAC_{c},\ Coordinator_{j}\ \big)\Big)$

- S32:

  $\square\Big(aborted^{t} \ \in RFQ \wedge aborted^{t}\ .IsDistributed \Rightarrow$

  $\lozenge\ send\big(aborted^{t}\ ,\ RAC_{c},\ Coordinator_{j}\ \big)\Big)$

Proof. Property S26 indicates that if the SAQ queues a Prepare To Commit message for which there is no corresponding Ready To Commit or Refuse To Commit response in either the RFQ or the RRQ, then RAC_{c} eventually sends $Aborted^{t}$ to c when it is recovered (see chapter 3). Furthermore, all such transactions must be recovered and restarted at c according to properties P13-P16. Property S27 indicates that if the SRQ queues a Commit message for which there is no corresponding Committed response in either the RFQ or the RRQ, then RAC_c eventually sends Commit to the recovered c. Property S28 indicates that if the SRQ queues an Abort message for which there is no corresponding Aborted response in either the RFQ or the RRQ, then RAC_c eventually sends Abort to the recovered c.

Properties 29-32 ensure that for distributed transactions, the coordinator of the distributed transaction receives the last response that RAC_c received from c.

To ensure these properties, RAC_c determines the recovery actions for each failed transaction according the algorithm shown in fig 9.6. First, RAC_c determines recovery actions of failed transactions that reached the second phase of the 2PC protocol. To accomplish this, RAC_c iterates over Commit messages queued in the SRQ. For each such transaction such that there is no corresponding Committed response in either the RFQ or the RRQ, then the action is to resend the Commit message to c when it is recovered. Otherwise, if there is a Committed response such that this transaction is distributed, then RAC_c sends Committed to the coordinator of the distributed transactions.

Similarly, RAC_c iterates over Abort messages queued in the SRQ. For each such transaction such that there is no corresponding Aborted response in either the RFQ or the RRQ, then the action is to resend the Abort message to c when it is recovered. Otherwise, if there is an Aborted response such that this transaction is distributed, then RAC_c sends Aborted to the coordinator of the distributed transactions.

Finally, RAC_s determines recovery actions for transactions that failed during the first phase of 2PC. To accomplish this, RAC_s iterate over the Prepare To Commit requests queued in the SAQ. For each failed transaction in the SAQ, if there is no Ready To

Commit or Refuse To Commit response queued in either the RFQ or the RRQ, then RAC_s must sends the Abort message to c when it is recovered and then restart this transaction at c. Otherwise if a response is found, then the action is to send this response to the coordinator of the distributed transaction if this transaction is distributed or send Commit or Abort to c if this transaction is non-distributed.

```
actions ← { }
for each $\text{Commit}^t \in \text{SRQ}$ then
    If $t \in \text{failed} \land \text{Committed}^t \notin RFQ \cup RRQ$ then
        actions ← actions ∪ {$\text{Commit}^t$ }
    Else if $t \in \text{failed} \land \text{Committed}^t \in RFQ \cup RRQ \land$
$\text{Committed}^t$.isDistributed then
        actions ← actions ∪ {$\text{Committed}^t$ }
    end
end
for each $\text{Abort}^t \in \text{SRQ}$ then
    If $t \in \text{failed} \land \text{Aborted}^t \notin RFQ \cup RRQ$ then
        actions ← actions ∪ {$\text{Abort}^t$ }
    Else if $t \in \text{failed} \land \text{Aborted}^t \in RFQ \cup RRQ \land \text{Aborted}^t$.isDistributed
     then
```

```
            actions $\leftarrow$ actions $\cup$ {$\text{Aborted}^{t}$ }
        **end**
**end**
**for each** $\text{PrepareToCommit}^{t} \in \text{SAQ} \wedge (\text{Abort}^{t} \vee \text{Commit}^{t}) \notin \text{SRQ}$ **then**
        **If** $t \in \text{failed} \wedge \left(\text{ReadyToCommit}^{t} \vee \text{RefuseToCommit}^{t}\right) \notin RFQ \cup RRQ$
         **then**
            actions $\leftarrow$ actions $\cup$ {$\text{Abort}^{t}$ , $\text{Restart}^{t}$ }
        **Else If**
$t \in$
$\text{failed} \wedge \left(\text{ReadyToCommit}^{t}\right) \in$
$RFQ \cup RRQ \wedge \sim\text{ReadyToCommit}^{t}.isDistributed$ **then**
            actions $\leftarrow$ actions $\cup$ {$\text{Commit}^{t}$ }
        **Else If**
$t \in$
$\text{failed} \wedge \left(\text{RefuseToCommit}^{t}\right) \in RFQ \cup RRQ \wedge \sim\text{RefuseToCommit}^{t}.isDistributed$
**then**
            actions $\leftarrow$ actions $\cup$ {$\text{Abort}^{t}$ }
        **Else If**
$t \in$
$\text{failed} \wedge \left(\text{ReadyToCommit}^{t}\right) \in RFQ \cup RRQ \wedge \text{ReadyToCommit}^{t}.isDistributed$
**then**
```

```
                actions $\leftarrow$ actions $\cup$ {ReadyToCommit$^t$ }
        **Else If**
t $\in$
failed $\wedge$ $\left(\text{RefuseToCommit}^t\right) \in RFQ \cup RRQ \wedge \text{RefuseToCommit}^t.isDistributed$
**then**
                actions $\leftarrow$ actions $\cup$ {RefuseToCommit$^t$ }
        **end**
**end**
```
Figure 9.6 algorithm executed by the RAC to recover failed 2PC transactions

9.4.3 Dynamic Adaptation

This section defines several properties that are provided by RAC_c during adaptation of c.

Properties S33, S34: *when RAC_c receives the Passivate command from RAM_i, then RAC_c eventually transitions to the Quiescent state. Furthermore, RAC_c does not transition to the Quiescent state unless the service is not engaged in any transactions:*

$$\square(receive(passivate, RAC_c,\ RAM_i) \Rightarrow \lozenge\ RAC_c.state = Quiescent\ )$$

$$\square\left((RAC_c \cong Quiescent\ ) \cup RAC_c.\text{ActiveTransactionsCount} = 0)\right)$$

<u>Proof.</u> From the design of the state machine executed by Connector Control of RAC_c (see figure 3.6 in chapter 3), Connector Control maintains the number of active transactions that c is currently engaged according to the following rules:

- $send(reqeust_1^t, RAC_c, c)\ \rightarrow\ increment(ActiveTransactionCount)$

- $receive\left(committed^{t} \vee aborted^{t}, RAC_{c}, c\right) \rightarrow$
  $decrement(ActiveTransactionCount)$

By design, if Connector Control of RAC_c receives the Passivate command while $ActiveTransactionCount = 0$, then Connector Control transitions immediately to the *Quiescent* state (see fig. 3.6 in chapter 3). Otherwise, Connector Control transitions to the intermediate $Passivating$ state in which it allows existing transactions to terminate normally but does not forward any new transactions to c according to the following property:

- S35: $\square\ \big(RequestCoordinator_{c}.state =$
  $Passivating\ \wedge receive\left(request_{1}^{t}, RAC_{c},\ Client_{j}\right) \rightarrow$
  $queue(request_{1}^{t}, RequestCoordinator_{c}, SPQ)\ \wedge \sim$
  $send(request_{1}^{t}, RequestCoordinator_{c}, ConnectorControl_{c})\big)$

This property is ensured by the state machine executed by $RequestCoordinator_c$ since the action for receiving requests that initiate new transactions with c is to hold the request in the SPQ. Thus, this property ensures that eventually $ActiveTransactionCount = 0$ and that RAC_c transitions from the $Passivating$ to the *Quiescent* state.

Properties S36: *while in the Quiescent state, RAC_c ceases forwarding all requests it receives from clients to c and holds these requests in the SPQ. Formally:*

$$\square\left(receive\left(reqeust_1^t, RAC_c, Client_j\right)\ \&\&\ RAC_c.state = Quiescent\ \rightarrow\right.$$

$$\left.\sim send(reqeust_1^t, RAC_c, c)\ \wedge queue(request_i^t, RequestCoordinator_c, SPQ)\right)$$

<u>*Proof.*</u> This property is satisfied by the design of the Request Coordinator as can be clearly seen from the state machine executed by this coordinator (c.f. fig. 3.4 in chapter 3). In this state machine, when the Request Coordinator transitions eventually to the Quiescent state according to property P34, then the action while in the Quiescent state for receiving client requests is to hold these requests into the SPQ.

Properties S37: *let $adapted(c)$ denotes completion of adaptation of stateful component c. When c adaptation is completed, then RAC_c is eventually reactivated. Formally:*

$$\square(adapted(c)\ \Rightarrow \lozenge\ receive(reactivate, RAC_c,\ RAM_i)\ )$$

<u>*Proof*</u>. This property is satisfied by design since when the Recovery and Adaptation Manager RAM_i finishes adapting c, the RAM always reactive the input RAC of the component affected by adaptation (see section 9.2).

Properties S38: *when RAC_c is reactivated after c is adapted to c', then RAC_c resumes sending messages, including held message, to c'. Formally, $\forall request_i^t\ \in SPQ$:*

$$\square(receive(reactivate, RAC_c, \quad RAM_i) \rightarrow \lozenge\ send(request_i^t, RequestCoordinator_c, \quad ConnectorControl_c))$$

This property indicates that when RAC_c is reactivated, then eventually $RequestCoordinator_c$ forwards all requests queued in its SPQ to $ConnectorControl_c$, including all requests held by $RequestCoordinator_c$ (according to properties S35 and S36). This property is ensured by design as shown in the state machine executed by the Request Coordinator (see fig. 3.4 in chapter 3) since when this coordinator is reactivated, it sends all messages queued in the SPQ to the Connector Control. Connector Control then forwards these requests to c' normally according to property S4.

9.5 Recovery and Adaptation Properties of RAC for Asynchronous Patterns

This section describes several properties achieved by the Recovery and Adaptation Connector (RAC) for recovering and adapting consumers in asynchronous patterns during normal execution, failure recovery, and dynamic adaptation. The goal of a RAC_c that handles recovery and adaptation concerns of a consumer c is to ensure the following property:

Property C1: *if RAC_c receives a producer message, then eventually RAC_c receives an acknowledgement from the consumer indicating that it is done with the producer's message. Formally:*

$$\Box\left(receive\left(message^{t}\,,\ RAC_c,\ Producer_j\right)\ \Rightarrow \Diamond\, receive\left(ACK^{t}\,, RAC_c,\ c\right)\right)$$

9.5.1 Normal Execution

This section defines several properties of RAC_c during normal operation (i.e. assuming that there are no failures or adaptation) which ensure that all asynchronous transactions received by the RAC are maintained in queues until the consumer acknowledges completion of these transactions.

Property C2: *any received message by RAC_c from a producer is always queued by RAC_c into the Service Pending Queue (SPQ). Formally:*

$$\Box\left(receive\left(message^t\,,\ RAC_c,\ Producer_j\right) \rightarrow queue\left(message^t\,, RAC_c,\ SPQ\right)\right)$$

<u>*Proof.*</u> This property is satisfied by design of the state machine executed by the Queue Coordinator of RAC_c since the *first* action executed by this coordinator whenever it receives a message from a producer is to queue that request in the SPQ (c.f. fig. 4.4 in chapter 4).

Property C3: *if RAC_c queued $message^t$ into the SPQ, then RAC_c will eventually forward this message to consumer c. Formally:*

$$\Box\left(queue\left(message^t\,,\ RAC_c,\ SPQ\right)\ \Rightarrow \Diamond\ send\left(message^t\,, RAC_c,\ c\right)\right)$$

<u>*Proof.*</u> By design, when the Queue Coordinator of RAC_c receives $message^t$, then in addition to queueing the message into the SPQ, the action is to also forward this message to Connector Control of RAC_c (c.f. fig. 4.4 in chapter 4). Formally, $receive\left(message^t\,,\ QueueCoordinator_c,\ Client_j\right) \rightarrow send\left(message^t\,,\ QueueCoordinator_c,\ ConnectorControl_c\right)$. From the state machine executed by the Connector Control (c.f. fig. 4.3 in chapter 4), when Connector Control receives $message^t$ then the actions are to

(1) forward the message to c and (2) instruct the Queue Coordinator to move this message from the SPQ to the Service Active Queue (SAQ), i.e., $receive\left(message^{t}\ , ConnectorControl_{c},\ QueueCoordinator_{c}\right) \rightarrow send\left(message^{t}\ ,\ ConnectorControl_{c},\ c\ \right) \wedge move\left(message^{t}\ ,\ QueueCoordinator_{c},\ SPQ, SAQ\right)$.

Property C4: *when RAC_c forwards $message^t$ to consumer c, then RAC_c moves this message from the SPQ to the Service Active Queue (SAQ), indicating that this message is currently being processed by c. Formally:*

$$\square\left(send\left(message^{t}\ ,\ RAC_{c},\ c\right) \rightarrow\ move\left(message^{t}\ ,\ RAC_{c},\ SPQ, SAQ\right)\right)$$

<u>*Proof.*</u> By design: $receive\left(message^{t}\ ,\ ConnectorControl_{c},\ QueueCoordinator_{c}\right) \rightarrow send\left(message^{t}\ ,\ ConnectorControl_{c},\ c\ \right) \wedge\ send\left(message^{t}\ ,\ ConnectorControl_{c},\ QueueCoordinator_{c}\right)$.

That is, receiving a producer message causes Connector Control to send the message to the consumer c (as shown in property C3) and also to send this message to the Queue Coordinator. By design, sending the producer's message back to the Queue Coordinator causes the coordinator to move the producer's message from the SPQ to the SAQ. I.e. $receive\left(message^{t}\ ,\ QueueCoordinator_{c},\ ConnectorConnector_{c}\right) \rightarrow move\left(message^{t}\ ,\ QueueCoordinator_{c},\ SPQ, SAQ\right)$.

Properties C5: RAC_c does not remove any messages from the SAQ unless it receives an acknowledgement from the consumer indicating that it is done with the producer's message. *Formally:*

$$\square\left(\sim dequeue\left(message^t\ ,\ RAC_c, SAQ\right)\ \cup receive(ACK^t,\ \ RAC_c,\ c\ \ )\right)$$

<u>*Proof*</u>. From properties C4, RAC_c eventually forwards the messages it receives from producers to consumer c. Since by assumption the consumer c is running normally, then c eventually sends to RAC_c an ACK for each producer message it receives from RAC_c according to the following sequence of events:

- Sequence 1: $send\left(message^t\ , RAC_c,\ c\right) \rightarrow receive\left(message^t\ , c,\right.$ $\left.RAC_c\right) \rightarrow send\left(ACK^t\ , c, RAC_c\right)\ \rightarrow receive\left(ACK^t\ , RAC_c,\ c\right)$.

By design in the message sequence executed by Connector Control of RAC_c (see fig. 4.3 in chapter 4), receiving an ACK from the consumer causes Connector Control to instruct the Queue Coordinator to remove the message of this transaction from the SAQ.

9.5.2 Failure Recovery

This section defines several properties that are provided by RAC_c during failure recovery of consumer c. These properties ensure that any asynchronous transaction that failed due to failure of c is eventually recovered and forwarded when c is recovered. Furthermore, these properties ensure that all transactions received by the RAC while c is in the failed state are queued until c is recovers from failure.

Properties C6: Let $fail(c)$ indicates a failure event occurring at consumer c.

when consumer c fails, then eventually RAC_c is notified of c failure by the Recovery and Adaptation Manager RAM_i. *Formally:*

$$\Box\big(fail(c) \Rightarrow \lozenge\, receive(Failed_c, RAC_c,\ RAM_i)\big)$$

<u>*Proof*</u>. This property is satisfied by design since while determining the recovery plan, RAM_i always notifies the input RAC of component failure as shown previously in section 9.2.

Property C7: when RAC_c is notified by RAM_i of c failure, then eventually RAC_c ceases forwarding messages to c and holds all messages to c in the SPQ. *Formally:*

$$\Box\big(QueueCoordinator_c.state = failed \ \wedge receive\big(message^t\ , RAC_c,\ Producer_j\big) \rightarrow queue\big(message^t\ , QueueCoordinator_c, SPQ\big)\ \wedge \sim send(message^t\ , QueueCoordinator_c, ConnectorControl_c)\big)$$

<u>*Proof.*</u> This property is satisfied by the design of the Queue Coordinator as can be clearly seen from the state machine executed by this coordinator (c.f. fig. 4.4 in chapter 4). In this state machine, when the coordinator is notified of c failure according to property C6, then the Queue Coordinator transitions into the Failed state in which the action for receiving producer messages is to hold these requests into the SPQ. Therefore, all messages received by RAC_c after it has been notified of c failure are held in the SPQ.

Property C8: Let $tFail(t)$ denotes a failure event occurring to transaction t. RAC_c is capable of identifying a transaction t as *failed* according to the following property:

$$\square\Big(tFail(t) \Rightarrow send\big(message^{t}\ ,\ \ RAC_c,\ c\big) \land \sim receive\big(ACK^{t}\ ,\ \ RAC_c,\ c\big)\Big)$$

Proof. This property indicates that a transaction t is considered failed if RAC_c forwarded a producer message of a transaction t to c but did not receive an ACK for this transaction from c (due to c failure). By design, RAC_c is capable of determining all failed transactions as follows. From Property C4, RAC_c moves all asynchronous messages it forwards to c from the SPQ to the SAQ. From property C5, RAC_c does not remove any asynchronous messages from the SAQ unless it receives an ACK from the consumer that it is done with the producer's message. Therefore, all asynchronous messages queued in the SAQ are of failed transactions, which satisfies property C8.

Property C9: RAC_c recovers all identified failed transactions by moving these transactions from the SAQ to the SPQ according to the following property.

$\forall \text{message}^{\,t}\ \ \in SAQ\ s.t.t\ has\ failed$:

$$\square\big(\lozenge\ move(\text{message}^{\,t}\ , RAC_c, SAQ, SPQ)\big)$$

Proof. Property C9 indicates that RAC_c always eventually move messages of every failed asynchronous transaction from the SAQ to the head of the SPQ. To ensure these properties, RAC_c recovers failed asynchronous transactions by iterating over the asynchronous messages queued in the SAQ. For each message, RAC_c moves the message from the SAQ to the SPQ.

Property C10: let $recovered(c, n)$ indicate recovery of consumer c to node n. When component c is recovered, RAC_c is eventually reactivated. Formally:

$$\square(recovered(c, n) \Rightarrow \lozenge\, receive(reactivate, RAC_c,\ RAM_i)\ )$$

Proof. This property is satisfied by design since while executing the recovery plan, the Recovery and Adaptation Manager RAM_i always notifies the input RAC of the failed component of component recovery as shown previously in section 9.2.

Properties C11: when RAC_c is reactivated after c is recovered (as explained in section 9.2), then RAC_c resumes sending messages, including held and lost message, to c. Formally, $\forall message^t \in$ SPQ:

$$\square\big(receive(reactivate, RAC_c, \quad RAM_i) \Rightarrow \lozenge\, send(message^t\ ,$$
$$QueueCoordinator_c,\quad ConnectorControl_c)\big)$$

Proof. This property indicates that when RAC_c is reactivated, then eventually $QueueCoordinator_c$ forwards all messages queued in its SPQ to $ConnectorControl_c$, including all messages held by $QueueCoordinator_c$ (according to property C7) and recovered messages (according to property C9). This property is ensured by design as shown in the state machine executed by this coordinator (see fig. 4.4 in chapter 4) since when this coordinator is reactivated, it sends all messages queued in the SPQ to the Connector Control. Connector Control then forwards these requests to c normally according to property C3.

9.5.3 Dynamic Adaptation

This section defines several properties that are provided by RAC_c during adaptation of consumer c. These properties ensure that c can be adapted only if c has completed all transactions that it is currently engaged in and will not receive any new transactions from other components. Furthermore, these properties ensure that any new transactions received by the RAC while the component is being adapted are queued until dynamic adaptation is completed.

Properties C12: when RAC_c receives the Passivate command from RAM_i, then RAC_c eventually transitions to the Quiescent state. Furthermore, RAC_c does not transition to the quiescent state unless the consumer is not engaged in any transactions. Formally:

$$\Box(receive(passivate, RAC_c,\ RAM_i) \Rightarrow \Diamond\, RAC_c.state = quiescent)$$

$$\Box\Big((RAC_c.state \cong \text{ Quiescent})\ \cup RAC_c.\text{ActiveTransactionsCount} = 0)\Big)$$

<u>*Proof.*</u> From the design of the state machine executed by Connector Control of RAC_c (see figure 4.3 in chapter 4), Connector Control maintains the number of active transactions that c is currently engaged according to the following rules:

- $send\big(message^t\ , RAC_c, c\big)\ \rightarrow\ increment(ActiveTransactionCount)$
- $receive\big(ACK^t\ , RAC_c, c\big)\ \rightarrow\ decrement(ActiveTransactionCount)$

By design, if Connector Control of RAC_c receives the $passivate$ command while $ActiveTransactionCount = 0$, then Connector Control transitions immediately to the $quiescent$ state (see fig. 4.3 in chapter 4). Otherwise, Connector Control transitions to the intermediate $passivating$ state in which it allows existing transactions to terminate normally but does not forward any new transactions to c according to the following property:

- P20: $\square\ (QueueCoordinator_c.state =$ $passivating\ \land receive(message^t\ , RAC_c,\ Producer_j) \rightarrow$ $queue(message^t\ , QueueCoordinator_c, SPQ)\ \land \sim$ $send(message^t\ , QueueCoordinator_c, ConnectorControl_c))$

This property is ensured by the state machine executed by $QueueCoordinator_c$ since the action for receiving asynchronous messages that initiate new transaction with c is to hold the message in the SPQ. This property ensures that eventually $ActiveTransactionCount = 0$ and that RAC_c transitions from the $passivating$ to the $quiescent$ state.

Properties C13: while in the Quiescent state, RAC_c ceases forwarding all asynchronous messages it receives from producers to c and holds these message in the SPQ. Formally:

$$\square\ \left(receive(message^t\ , RAC_c, Producer_j)\ \land RAC_c.state = quiescent\ \rightarrow\right.$$
$$\left.\sim send(message^t\ , RAC_c, c)\ \land queue(message^t\ , QueueCoordinator_c, SPQ)\right)$$

<u>*Proof.*</u> This property is satisfied by the design of the Queue Coordinator as can be clearly seen from the state machine executed by this coordinator (c.f. fig. 4.4 in chapter 4). In this state machine, when the Queue Coordinator transitions eventually to the Quiescent state according to property C12, then the action while in the Quiescent state for receiving asynchronous message is to hold these messages into the SPQ.

Properties P22: let $adapted(c)$ denotes completion of consumer c adaptation. When c adaptation is completed, then RAC_c eventually is reactivated. Formally:

$$\Box(adapted(c) \ \Rightarrow \Diamond\, receive(reactivate, RAC_c,\ RAM_i)\ )$$

Proof. This property is satisfied by design since when the Recovery and Adaptation Manager RAM_i finishes adapting c, the RAM always reactive the input RAC of the component affected by adaptation (see section 9.2).

Properties P23: when RAC_c is reactivated after c is adapted to c', then RAC_c resumes sending asynchronous messages, including held asynchronous message, to c'. Formally, $\forall message^t \ \in$ SPQ:

$$\Box\big(receive(reactivate, RAC_c, \quad RAM_i) \Rightarrow \Diamond\, send(message^t\ , \\ QueueCoordinator_c, \quad ConnectorControl_c)\big)$$

This property indicates that when RAC_c is reactivated, then eventually $QueueCoordinator_c$ forwards all requests queued in its SPQ to $ConnectorControl_c$, including all requests held by $QueueCoordinator_c$ (according to properties C20 and C21). This property is ensured by design as shown in the state machine executed by the Queue Coordinator (see fig. 4.4 in chapter 4) since when this coordinator is reactivated, it sends all messages queued in the SPQ to the Connector Control. Connector Control then forwards these requests to c' normally according to property C3.

10 EXPERIMENTAL DESIGN AND VALIDATION

I conducted detailed experiments of self-healing and self-configuration scenarios to evaluate the approach described in this dissertation. To carry out the experiments, I implemented the DARE framework as well as the architectures of two case studies: an Online Shopping System and an Emergency Monitoring System (Gomaa, 2011). In these experiments, each component and RAC was implemented in Java and has a separate thread of control. In addition, Java Sockets were used for message delivery. The implemented architecture runs on a cluster consisting of 30 nodes. Thus, components and RACs are both concurrent and distributed in these experiments. Section 10.1 describes the various experiments in this chapter. Sections 10.2-10.7 provide the details of each experiment.

10.1 Experimental Design

In order to validate the approach in this dissertation, I conducted several experiments as follows:

- Experiment 1: this experiment validates the design of the service RAC used for *stateless* services which does not maintain state information about its clients (c.f. section 3.1). This experiment is described in section 10.2.1.

- Experiment 2: this experiment validates the design of the service RAC used for *stateful* services with *non-idempotent operations* (c.f. section 3.2). This experiment is described in section 10.2.2.
- Experiment 3: this experiment validates the design of the service RAC used for *stateful* services that participate in *distributed transactions* (c.f. section 3.3.5). This experiment is described in section 10.2.3.
- Experiment 4: this experiment validates the design of the coordinator RAC used for *coordinators* in service-oriented architectures (c.f. section 3.3.4). This experiment is described in section 10.2.4.
- Experiment 5: this experiment validates the design of the consumer RAC used for consumers in *asynchronous* patterns (c.f. chapter 4). This experiment is described in section 10.3.
- Experiment 6: this experiment validates the design of the DARE framework (c.f. chapter 5). This experiment is described in section 10.4.
- Experiment 7: this experiment validates the design of the Assistant Recovery and Adaptation Connector (c.f. chapter 6). This experiment is described in section 10.5.
- Experiment 8: this experiment validates the capability of the RAC to recover from run-time failures using message logging (c.f. chapter 7). This experiment is described in section 10.6.

- Experiment 9: this experiment validates the design of the reusable RAC which is capable of integrating multiple patterns (c.f. chapter 8). This experiment is described in section 10.7.

10.2 Experimentation with RAC in SOA Patterns

This section describes the experimental design and results of the RAC used in SOA patterns (c.f. chapter 3). This RAC contains the following queues for maintaining client requests and service responses:

- Service Pending Queue (SPQ): this queue stores client requests received by the RAC but that have not yet been forwarded to the service.
- Service Active Queue (SAQ): this queue stores client requests that have been forwarded to the service but do not have corresponding service responses at the RAC.
- Service Recovery Queue (SRQ): this queue stores client requests that have corresponding service responses at the RAC.
- Response Forwarding Queue (RRQ): this queue stores responses that are not forwarded to the requesting clients.
- Response Recovery Queue (RFQ): this queue stores responses that have been forwarded to requesting clients.

The goal of the experiments is to validate recovery and adaptation capability of the RAC without losing state information. In order to carry out experimentation, I implemented the architecture of the Online Shopping System case study (Gomaa, 2011), which is an example of a service-oriented architecture. In this case study, customers can request to purchase items from suppliers (see Figure 10.1). Several services are involved to carry out purchase requests such as the Customer Account Service, Delivery Order Service, Catalog Service, and Credit Card Service. Therefore, coordinators are used to facilitate integration of these services. Thus, this case study helps with experimenting with the service RAC as it involves the SOA pattern described previously in chapter 3.

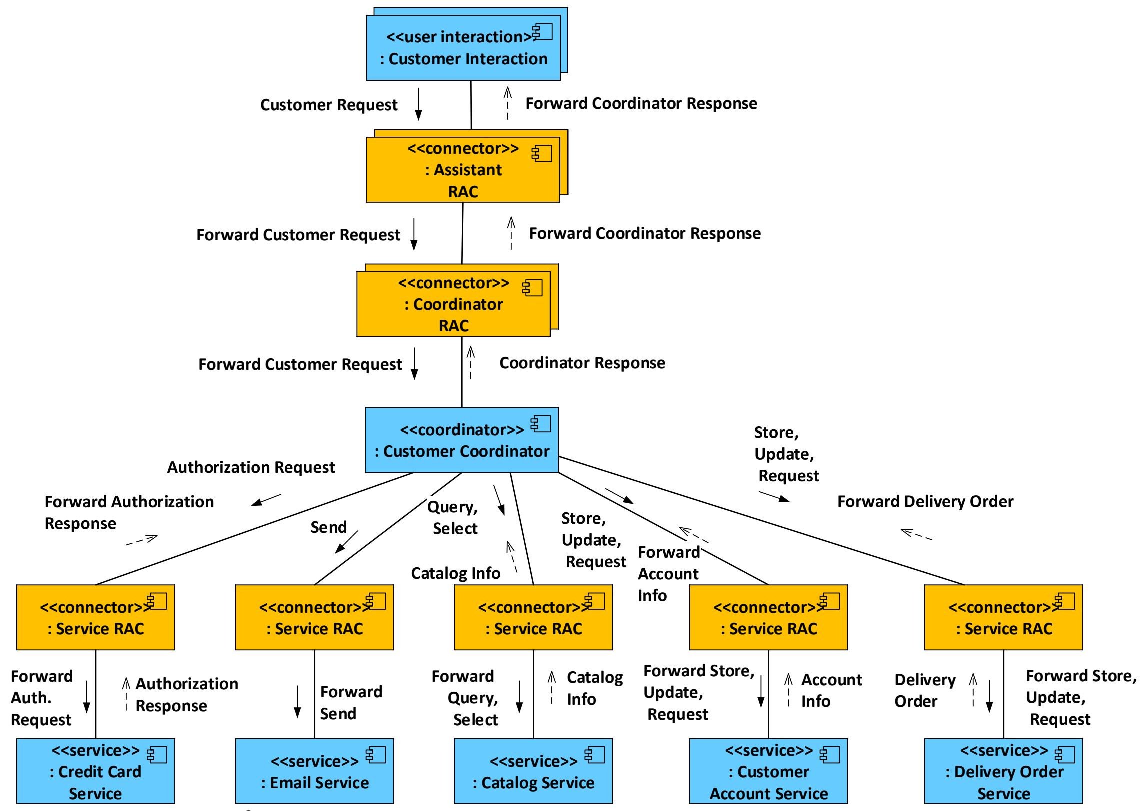


Figure 10.1 Fragment of Online Shopping System case study (Gomaa, 2011)

10.2.1 Experimentation with RAC for Stateless Services

This section describes the validation of the RAC that handles adaptation and recovery concerns of *stateless* services which do not maintain state information about their clients (see section 3.1 in chapter 3).

Self-Healing Scenario

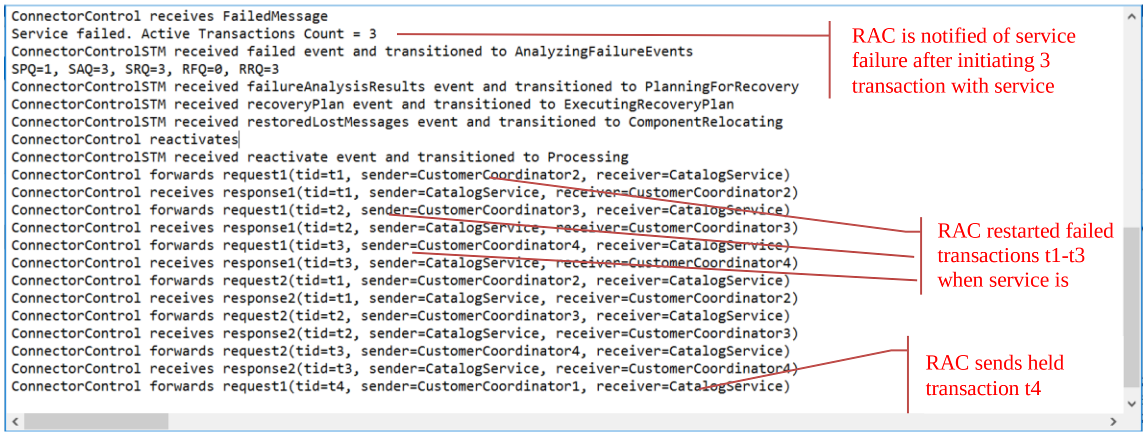
The service failure scenario demonstrates the ability of the service RAC to recover failed transactions. In this scenario, the Catalog Service (CS) is concurrently processing three active transactions (t1-t3) at the time of failure. As a result, the expected behavior in this scenario includes:

1. The CS RAC receives a failure notification from the Recovery and Adaptation Manager (RAM).
2. The CS RAC determines a recovery action to recover the failed transactions t1-t3.
3. The CS RAC receives the Reactivate command from the Recovery and Adaptation Manager to resume sending messages to the recovered CS. As a result, CS RAC resends transactions t1-t3 to the recovered CS.

In this scenario, the execution trace (Figure 10.2) indicates that the content of the RAC's queues is as follows:

- The SPQ holds one request due to service failure:
    - Request1(t4, CC1, CS), where *t4* is the identifier of the transaction, *CC1* is the identifier of the message sender, and *CS* is the identifier of the message recipient.
- The SAQ contains three requests that have been forwarded to the service as follows:

  - Request2(t1, CC2, CS)
  - Request2(t2, CC3, CS)
  - Request2(t3, CC4, CS)
- The SRQ contains three requests:
  - Request1(t1, CC2, CS)
  - Request1(t2, CC3, CS)
  - Request1(t3, CC4, CS)
- The RFQ does not contain any queued responses.
- The RRQ contains three forwarded responses:
  - Response1(t1, CS, CC2)
  - Response1(t2, CS, CC3)
  - Response1(t3, CS, CC4)



```
ConnectorControl receives FailedMessage
Service failed. Active Transactions Count = 3
ConnectorControlSTM received failed event and transitioned to AnalyzingFailureEvents
SPQ=1, SAQ=3, SRQ=3, RFQ=0, RRQ=3
ConnectorControlSTM received failureAnalysisResults event and transitioned to PlanningForRecovery
ConnectorControlSTM received recoveryPlan event and transitioned to ExecutingRecoveryPlan
ConnectorControlSTM received restoredLostMessages event and transitioned to ComponentRelocating
ConnectorControl reactivates
ConnectorControlSTM received reactivate event and transitioned to Processing
ConnectorControl forwards request1(tid=t1, sender=CustomerCoordinator2, receiver=CatalogService)
ConnectorControl receives response1(tid=t1, sender=CatalogService, receiver=CustomerCoordinator2)
ConnectorControl forwards request1(tid=t2, sender=CustomerCoordinator3, receiver=CatalogService)
ConnectorControl receives response1(tid=t2, sender=CatalogService, receiver=CustomerCoordinator3)
ConnectorControl forwards request1(tid=t3, sender=CustomerCoordinator4, receiver=CatalogService)
ConnectorControl receives response1(tid=t3, sender=CatalogService, receiver=CustomerCoordinator4)
ConnectorControl forwards request2(tid=t1, sender=CustomerCoordinator2, receiver=CatalogService)
ConnectorControl receives response2(tid=t1, sender=CatalogService, receiver=CustomerCoordinator2)
ConnectorControl forwards request2(tid=t2, sender=CustomerCoordinator3, receiver=CatalogService)
ConnectorControl receives response2(tid=t2, sender=CatalogService, receiver=CustomerCoordinator3)
ConnectorControl forwards request2(tid=t3, sender=CustomerCoordinator4, receiver=CatalogService)
ConnectorControl receives response2(tid=t3, sender=CatalogService, receiver=CustomerCoordinator4)
ConnectorControl forwards request1(tid=t4, sender=CustomerCoordinator1, receiver=CatalogService)
```

RAC is notified of service failure after initiating 3 transaction with service
RAC restarted failed transactions t1-t3 when service is
RAC sends held transaction t4

Figure 10.2 Fragment of the execution trace of the stateless service RAC during planned failure scenario

]

During the analysis activity which is handled in the Analyzing Failure Events state, the execution trace indicates that the service RAC determined that transactions *t1*, *t2*, and *t3* have failed, since these transactions have been initiated with the service before failure but service failure occurred before the service RAC received the responses that complete these transactions from the service. As a result, the execution trace indicates that the service RAC recovered requests of these transactions from the SAQ and the SRQ to the SPQ. The execution trace indicates that the state of the SPQ after moving these transactions are as follows:

- Request1(t1, CC2, CS)
- Request1(t2, CC3, CS)
- Request1(t3, CC4, CS)
- Request2(t1, CC2, CS),
- Request2(t2, CC3, CS)
- Request2(t3, CC4, CS)
- Request1(t4, CC1, CS)

When the RAC is reactivated after the CS has recovered, the execution trace indicates that the CS RAC restarted failed transactions t1-t3 with the recovered service and that these transactions eventually terminated normally. Furthermore, the trace indicates that the CS RAC forwarded transaction t4, which was held in the SPQ due to service failure, to the recovered service. Thus, the outcome of this experiment corresponds to what it is expected for this scenario.

Self-Configuration Scenario

To illustrate the behavior of the service RAC during adaptation of a stateless service, I use an adaptation scenario that involves adapting the Catalog Service (CS). In this scenario, the service is concurrently processing three transactions (t1-t3). As a result, the expected behavior in this scenario includes:

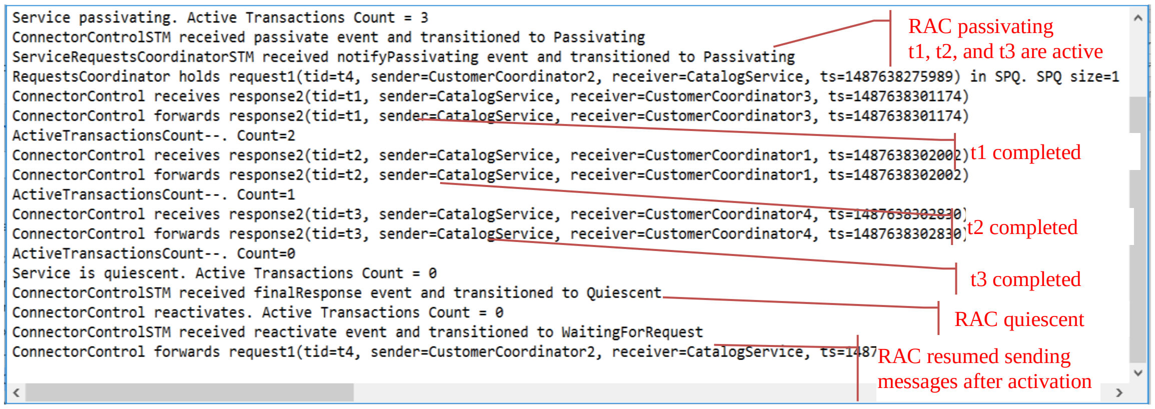
1. When the CS RAC receives the Passivate command from the RAM, the RAC transitions to the Passivating state in order to allow transactions t1-t3 to complete. Furthermore, the CS RAC holds any new transactions in the SPQ until dynamic adaptation is completed.
2. When transactions t1-t3 complete, the CS RAC transitions to the Quiescent state in which the CS can be safely adapted.
3. When dynamic service adaptation is completed, the CS RAC receives the Reactivate command. As a result, the RAC resumes sending messages, including held transactions in the SPQ to the adapted CS.

In this scenario, the contents of the various queues when the service RAC received the Passivate command are as follows:

- The SPQ does not contain requests.
- The SAQ contains three requests that have been forwarded to the service as follows:
    - Request2(t1, CC3, CS),
    - Request2(t2, CC1, CS)
    - Request2(t3, CC4, CS)
- The SRQ contains three requests:

    - Request1(t1, CC3, CS)
    - Request1(t2, CC1, CS)
    - Request1(t3, CC4, CS)
- The RFQ does not contain any queued responses.
- The RRQ contains three forwarded responses:
    - Response1(t1, CS, CC3)
    - Response1(t2, CS, CC1)
    - Response1(t3, CS, CC4)

As a result of passivation, the execution trace indicates that the service RAC transitioned into the Passivating state (Figure 10.3) where it permitted these three active transactions to terminate normally. During this state, the trace indicates that the RAC received a new transaction t4. As a result, the RAC held this transaction in the SPQ. After the service completed all active transactions, the execution trace indicates that the CS RAC transitioned to the Quiescent state at which time the service was dynamically replaced. After adaptation is completed, the service RAC received the reactivate command. As a result, the RAC transitioned to the Active state and forwarded the queued transaction t4 in the SPQ to the Catalog Service, which corresponds to the expected behavior for this scenario.



```
Service passivating. Active Transactions Count = 3
ConnectorControlSTM received passivate event and transitioned to Passivating
ServiceRequestsCoordinatorSTM received notifyPassivating event and transitioned to Passivating
RequestsCoordinator holds request1(tid=t4, sender=CustomerCoordinator2, receiver=CatalogService, ts=1487638275989) in SPQ. SPQ size=1
ConnectorControl receives response2(tid=t1, sender=CatalogService, receiver=CustomerCoordinator3, ts=1487638301174)
ConnectorControl forwards response2(tid=t1, sender=CatalogService, receiver=CustomerCoordinator3, ts=1487638301174)
ActiveTransactionsCount--. Count=2
ConnectorControl receives response2(tid=t2, sender=CatalogService, receiver=CustomerCoordinator1, ts=1487638302002)
ConnectorControl forwards response2(tid=t2, sender=CatalogService, receiver=CustomerCoordinator1, ts=1487638302002)
ActiveTransactionsCount--. Count=1
ConnectorControl receives response2(tid=t3, sender=CatalogService, receiver=CustomerCoordinator4, ts=1487638302830)
ConnectorControl forwards response2(tid=t3, sender=CatalogService, receiver=CustomerCoordinator4, ts=1487638302830)
ActiveTransactionsCount--. Count=0
Service is quiescent. Active Transactions Count = 0
ConnectorControlSTM received finalResponse event and transitioned to Quiescent
ConnectorControl reactivates. Active Transactions Count = 0
ConnectorControlSTM received reactivate event and transitioned to WaitingForRequest
ConnectorControl forwards request1(tid=t4, sender=CustomerCoordinator2, receiver=CatalogService, ts=1487
```

Figure 10.3 Fragment of the execution trace of stateless service RAC during planned adaptation scenario

10.2.2 Experimentation with RAC for Stateful Services with Non-Idempotent Operations

This section describes the validation of the RAC that handles adaptation and recovery concerns of *stateful* services which maintains persistent information about their clients (see section 3.2 in chapter 3).

Self-Healing Scenario

The service failure scenario demonstrates the ability of the service RAC to recover failed transactions (see section 3.2 in chapter 3). In this scenario, the Delivery Order Service (DOS) is concurrently processing four transactions (t1-t4), which are in different states at the time of failure, as follows:

- Transaction t1: this transaction fails after the DOS RAC sends the Prepare to Commit message to the service but before this RAC receives the Ready To Commit message from the service. Note that in this case, the service can fail either (1) before preparing to commit this transaction, (2) after preparing to commit the transaction but before sending Ready To Commit to the RAC, or (3) after sending Ready To Commit to the RAC such that this response was lost due to service failure. Although these three cases are not distinguishable from the RAC’s point of view, the RAC executes the same recovery actions to recover these cases.

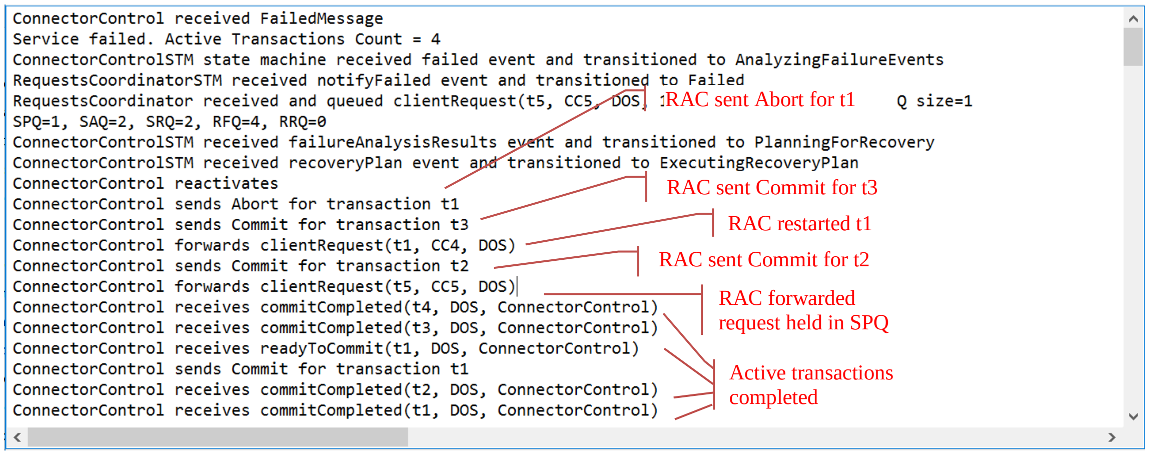
- Transaction t2: this transaction fails after the DOS RAC receives the Ready To Commit message from the service but before this RAC forwards Commit to the service.
- Transaction t3: this transaction fails after the DOS RAC sends Commit to the service but before the RAC receives Committed from the service. Therefore, the service can fail either (1) before committing this transaction, (2) after committing the transaction but before sending Committed to the RAC, or (3) after sending Committed to the RAC such that this response was lost due to service failure. Although these three cases are not distinguishable from the RAC’s point of view, the RAC executes the same recovery actions to recover these cases.
- Transaction t4: this transaction fails after the DOS RAC receives the Committed message from the service.

As a result, the expected behavior in this scenario includes:

1. The DOS RAC receives a failure notification from the Recovery and Adaptation Manager.
2. The DOS RAC determines the appropriate recovery actions to recover the failed transactions t1-t4 as follows: (1) send Abort for transaction t1 and then restart this

transaction with the service when it is recovered, (2) send Commit for transaction t2, and (3) resend Commit for transaction t3.

3. The DOS RAC receives the Reactivate command from the Recovery and Adaptation Manager to resume sending messages to the recovered DOS. As a result, the DOS RAC sends the appropriate recovery actions to the recovered DOS as shown in (2).



```
ConnectorControl received FailedMessage
Service failed. Active Transactions Count = 4
ConnectorControlSTM state machine received failed event and transitioned to AnalyzingFailureEvents
RequestsCoordinatorSTM received notifyFailed event and transitioned to Failed
RequestsCoordinator received and queued clientRequest(t5, CC5, DOS, 1            Q size=1
SPQ=1, SAQ=2, SRQ=2, RFQ=4, RRQ=0
ConnectorControlSTM received failureAnalysisResults event and transitioned to PlanningForRecovery
ConnectorControlSTM received recoveryPlan event and transitioned to ExecutingRecoveryPlan
ConnectorControl reactivates
ConnectorControl sends Abort for transaction t1
ConnectorControl sends Commit for transaction t3
ConnectorControl forwards clientRequest(t1, CC4, DOS)
ConnectorControl sends Commit for transaction t2
ConnectorControl forwards clientRequest(t5, CC5, DOS)
ConnectorControl receives commitCompleted(t4, DOS, ConnectorControl)
ConnectorControl receives commitCompleted(t3, DOS, ConnectorControl)
ConnectorControl receives readyToCommit(t1, DOS, ConnectorControl)
ConnectorControl sends Commit for transaction t1
ConnectorControl receives commitCompleted(t2, DOS, ConnectorControl)
ConnectorControl receives commitCompleted(t1, DOS, ConnectorControl)
```

Figure 10.4 Fragment of the execution trace of stateful service RAC during planned failure scenario

The execution trace (Figure 10.4) shows that after the RAC received a failure notification from the RAM, the RAC received a fifth transaction (t5) which was held in the SPQ. The content of the RAC's queues at this point is as follows:

- The SPQ contains one request that was held due to service failure:
  - Request(t5, CC5, DOS)
- The SAQ contains two prepare to commit requests that have been forwarded to the service:

    - Request(t2, CC1, DOS)
    - Request(t1, CC4, DOS)
- The SRQ contains two commit requests as follows:
    - Request(t4, CC2, DOS)
    - Request(t3, CC3, DOS)
- The RFQ contains four received responses:
    - ReadyToCommit(t2, DOS, CC1)
    - ReadyToCommit(t4, DOS, CC2)
    - Committed(t4, DOS, ConnectorControl)
    - ReadyToCommit(t3, DOS, CC3)
- RRQ does not indicate any responses that have been forwarded to clients.

During the analysis activity which is handled by the Analyzing Failure Events state, the execution trace indicates that the RAC determined a status for each of these active transactions as follows:

- For transaction t1, the RAC determined the status of this transaction as *Preparing*, since the SAQ contains the Prepare To Commit request to the service but neither the RFQ nor the RRQ contain the Ready To Commit response for this transaction.
- For transaction t2, the RAC determined the status of this transaction as *Prepared*, since the SAQ contains the Prepare To Commit request to the service and the RFQ contains the Ready To Commit response for this transaction.
- For transaction t3, the RAC determined the status of this transaction as *Committing*, since the SRQ contains the Commit request for this transaction but

neither the RFQ nor the RRQ queues contain the Committed response for this transaction.

- For transaction t4, the RAC determined the status of this transaction as *Committed*, since the SRQ contains the Commit request to the service for this transaction and the RFQ contains a Committed response for this transaction.

During the planning activity which is handled in the Planning for Recovery state, the RAC determined recovery actions for each active transaction as follows:

- For transaction t1, since this transaction failed while being prepared to commit by the service during the first phase of 2PC, the recovery actions determined by the RAC for this transaction after service recovery were (1) to abort this transaction with the service and then (2) to restart this transaction with the recovered service.
- For transaction t2, since this transaction failed after being prepared to commit by the service, the RAC determined a recovery action to send the Commit message for this transaction when the service has recovered.
- For transaction t3, since the service failed while committing this transaction, the recovery action determined by the RAC for this transaction was to resend the Commit message to the recovered service.
- Transaction t4 does not require any recovery actions since it was completed before service failure.

During the execution phase which is handled in the Executing Recovery Plan state, the service RAC recovered the requests of the transactions that must be restarted with the recovered service by moving these requests from the SAQ to the SPQ. In this scenario, only transaction t1 needs to be restarted with the recovered service. Therefore, the execution trace indicates that this message is moved from the SAQ to the SPQ. The content of SPQ after recovery is:

- Prepare(t1, CC4, DOS) // request that was recovered from the SAQ
- Prepare(t5, CC5, DOS) //request held in the SPQ due to service failure

When the DOS RAC is reactivated after the service has recovered, the execution trace indicates that the RAC aborted and then restarted transaction t1 with the recovered service, (2) requested the recovered service to commit transactions t2 and t3, and (3) forwarded transaction t5 which was previously held in the SPQ due to service failure. The execution trace indicates that service execution resumed normally and that all active transactions were eventually committed. Thus, the outcome of this experiment corresponds to what is expected for this scenario.

Self-Configuration Scenario

To illustrate the behavior of the service RAC during adaptation, I use an adaptation scenario that involves adapting the DOS. In this scenario, the DOS is concurrently processing four transactions, which are in different states at the time of adaptation, as described below:

- Transactions t1 and t2: the RAC receives the Passivate command after the Service RAC sends the Prepare To Commit messages for these transactions to the DOS but before the RAC receives the Ready To Commit responses from the service.

- Transactions t3 and t4: the RAC receives the Passivate command after the service RAC sends the Commit messages for these transactions to the DOS but before the RAC receives the Committed responses from the service for these transactions.

  As a result, the expected behavior in this scenario includes:

1. When the DOS RAC receives the Passivate command, the RAC transitions to the Passivating state in order to allow transactions t1-t4 to complete. Furthermore, the DOS RAC holds any new transactions in the SPQ until dynamic adaptation is completed.
2. When transactions t1-t4 complete, the DOS RAC transitions to the Quiescent state.
3. When dynamic service adaptation is completed, the DOS RAC receives the Reactivate command from the Recovery and Adaptation Manager. As a result, the RAC resumes sending messages to the adapted DOS including the transactions held in the SPQ.

When service adaptation is requested, the content of the RAC queues are as follows:

- The SPQ does not contain any requests held by the service RAC.
- The SAQ contains two prepare to commit requests sent to the service:
    - Request(t1, CC3, DOS)
    - Request(t2, CC2, DOS)
- The SRQ contains two commit requests sent to the service:
    - Request(t3, CC4, DOS)

    - Request(t4, CC1, DOS)
- The RFQ contains two service responses as follows:
    - ReadyToCommit(t3, DOS, CC4)
    - ReadyToCommit(t4, DOS, CC1)
- The RRQ does not indicate any responses that have been forwarded to clients.

As a result of passivation, the execution trace indicates that the service RAC transitioned into the Passivating state (Figure 10.5) where it permitted these four active transactions to gradually terminate. While in Passivating state, new transactions were received and queued by the Service RAC into the SPQ. After the service completed all active transactions, the execution trace indicates that the Service RAC transitioned to the Quiescent state at which time the service was dynamically replaced. During the adapting state, further requests are received and queued by RAC. After adaptation is completed, the service RAC received the reactivate command. As a result, the RAC transitioned to the Active state and forwarded all queued transactions in its SPQ to the service. At this point, normal execution is resumed between RAC and the service.

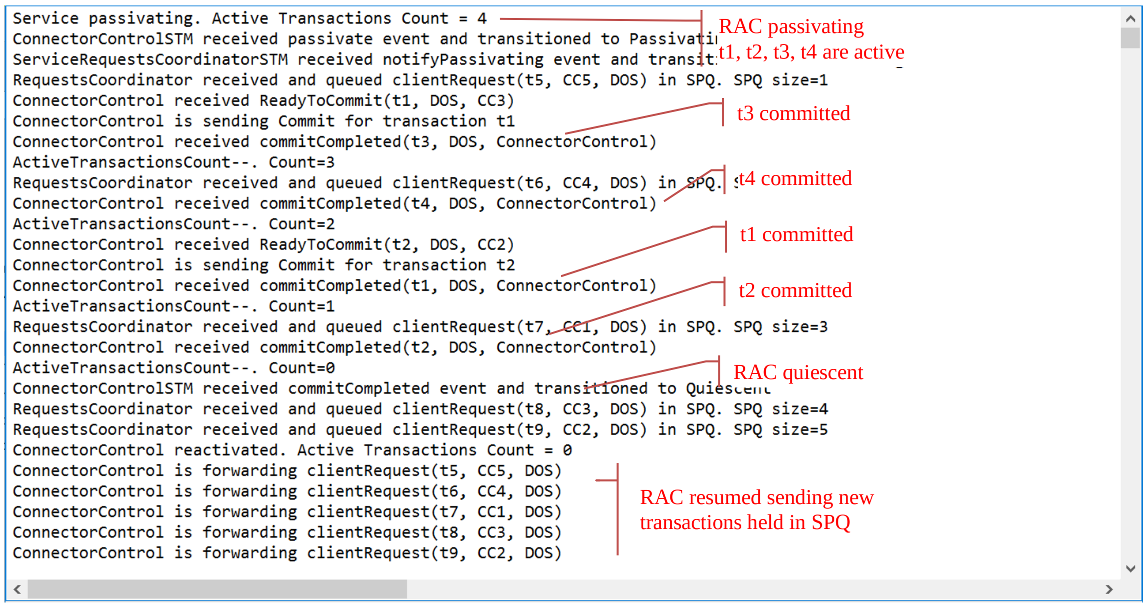

```
Service passivating. Active Transactions Count = 4
ConnectorControlSTM received passivate event and transitioned to Passivati
ServiceRequestsCoordinatorSTM received notifyPassivating event and transit
RequestsCoordinator received and queued clientRequest(t5, CC5, DOS) in SPQ. SPQ size=1
ConnectorControl received ReadyToCommit(t1, DOS, CC3)
ConnectorControl is sending Commit for transaction t1
ConnectorControl received commitCompleted(t3, DOS, ConnectorControl)
ActiveTransactionsCount--. Count=3
RequestsCoordinator received and queued clientRequest(t6, CC4, DOS) in SPQ.
ConnectorControl received commitCompleted(t4, DOS, ConnectorControl)
ActiveTransactionsCount--. Count=2
ConnectorControl received ReadyToCommit(t2, DOS, CC2)
ConnectorControl is sending Commit for transaction t2
ConnectorControl received commitCompleted(t1, DOS, ConnectorControl)
ActiveTransactionsCount--. Count=1
RequestsCoordinator received and queued clientRequest(t7, CC1, DOS) in SPQ. SPQ size=3
ConnectorControl received commitCompleted(t2, DOS, ConnectorControl)
ActiveTransactionsCount--. Count=0
ConnectorControlSTM received commitCompleted event and transitioned to Quiescent
RequestsCoordinator received and queued clientRequest(t8, CC3, DOS) in SPQ. SPQ size=4
RequestsCoordinator received and queued clientRequest(t9, CC2, DOS) in SPQ. SPQ size=5
ConnectorControl reactivated. Active Transactions Count = 0
ConnectorControl is forwarding clientRequest(t5, CC5, DOS)
ConnectorControl is forwarding clientRequest(t6, CC4, DOS)
ConnectorControl is forwarding clientRequest(t7, CC1, DOS)
ConnectorControl is forwarding clientRequest(t8, CC3, DOS)
ConnectorControl is forwarding clientRequest(t9, CC2, DOS)
```

Figure 10.5 Fragment of the execution trace of stateful service RAC during planned adaptation scenario

10.2.3 Experimentation with RAC for Distributed Transactions

This section describes the validation of the RAC that handle adaptation and recovery of services in the distributed transactions patterns.

Self-Healing Scenario

The service failure scenario demonstrates the ability of the service RAC to recover failed transactions. In this scenario, the Inventory Service (IS) is concurrently processing four transactions, which are in different states at the time of failure, as follows:

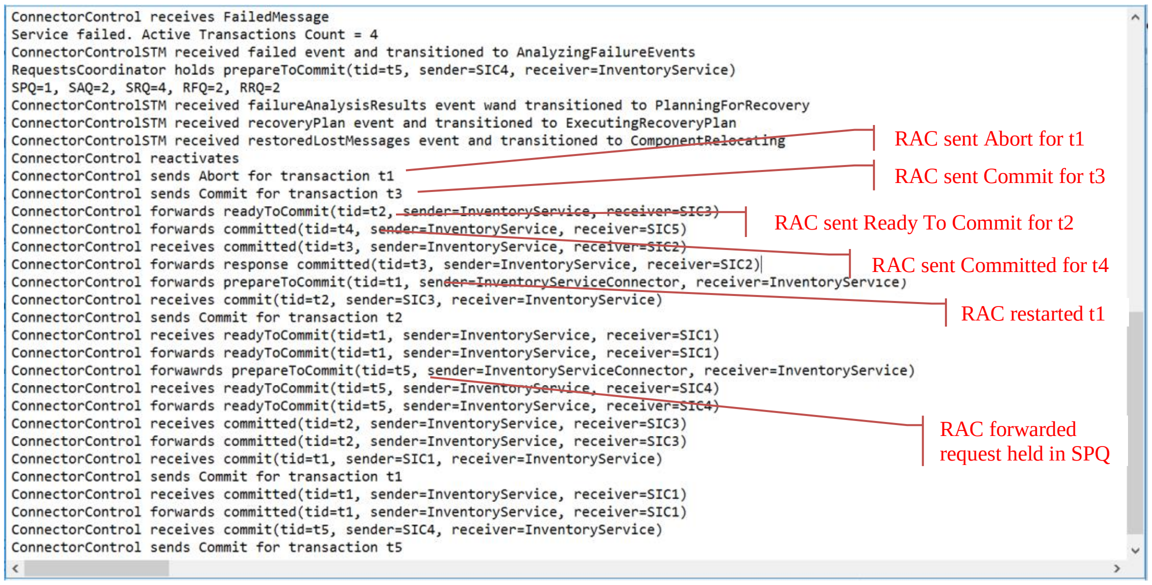
- Transaction t1: this transaction fails after the IS RAC sends the Prepare to Commit message to the service but before this RAC receives the Ready To Commit message from the service. Note that in this case, the service can fail either (1) before preparing to commit this transaction, (2) after preparing to commit the transaction but before sending Ready To Commit to the RAC, or (3) after sending Ready To Commit to the RAC such that this response was lost due to service failure. Although these three cases are not distinguishable from the RAC's point of view, the RAC executes the same recovery actions to recover these cases.

- Transaction t2: this transaction fails after the IS RAC receives the Ready To Commit message from the service but before this RAC forwards this response to the coordinator of the distributed transaction.
- Transaction t3: this transaction fails after IS RAC sends Commit to the service but before the RAC receives Committed from the service. Therefore, the service can fail either (1) before committing this transaction, (2) after committing the transaction but before sending Committed to the RAC, or (3) after sending Committed to the RAC such that this response was lost due to service failure. Although these three cases are not distinguishable from the RAC's point of view, the RAC executes the same recovery actions to recover these cases.
- Transaction t4: this transaction fails after the service RAC receives the Committed message from the service but before the RAC forwards this response to the coordinator of the distributed transaction.

As a result, the expected behavior in this scenario includes:

1. The IS RAC receives a failure notification from the Recovery and Adaptation Manager.

2. The IS RAC determines the appropriate recovery actions to recover the failed transactions t1-t4 as follows: (1) send Abort for transaction t1 and then restart this transaction with the service when it is recovered, (2) send Ready To Commit for transaction t2 to the coordinator of the distributed transaction, (3) resend Commit for transaction t3, and (4) send Committed for transaction t4 to the coordinator of this distributed transaction.
3. The IS RAC receives the Reactivate command from the Recovery and Adaptation Manager to resume sending messages to the recovered IS. As a result, IS RAC sends the appropriate recovery actions to the recovered IS as shown in (2).



```
ConnectorControl receives FailedMessage
Service failed. Active Transactions Count = 4
ConnectorControlSTM received failed event and transitioned to AnalyzingFailureEvents
RequestsCoordinator holds prepareToCommit(tid=t5, sender=SIC4, receiver=InventoryService)
SPQ=1, SAQ=2, SRQ=4, RFQ=2, RRQ=2
ConnectorControlSTM received failureAnalysisResults event wand transitioned to PlanningForRecovery
ConnectorControlSTM received recoveryPlan event and transitioned to ExecutingRecoveryPlan
ConnectorControlSTM received restoredLostMessages event and transitioned to ComponentRelocating
ConnectorControl reactivates
ConnectorControl sends Abort for transaction t1
ConnectorControl sends Commit for transaction t3
ConnectorControl forwards readyToCommit(tid=t2, sender=InventoryService, receiver=SIC3)
ConnectorControl forwards committed(tid=t4, sender=InventoryService, receiver=SIC5)
ConnectorControl receives committed(tid=t3, sender=InventoryService, receiver=SIC2)
ConnectorControl forwards response committed(tid=t3, sender=InventoryService, receiver=SIC2)
ConnectorControl forwards prepareToCommit(tid=t1, sender=InventoryServiceConnector, receiver=InventoryService)
ConnectorControl receives commit(tid=t2, sender=SIC3, receiver=InventoryService)
ConnectorControl sends Commit for transaction t2
ConnectorControl receives readyToCommit(tid=t1, sender=InventoryService, receiver=SIC1)
ConnectorControl forwards readyToCommit(tid=t1, sender=InventoryService, receiver=SIC1)
ConnectorControl forwawrds prepareToCommit(tid=t5, sender=InventoryServiceConnector, receiver=InventoryService)
ConnectorControl receives readyToCommit(tid=t5, sender=InventoryService, receiver=SIC4)
ConnectorControl forwards readyToCommit(tid=t5, sender=InventoryService, receiver=SIC4)
ConnectorControl receives committed(tid=t2, sender=InventoryService, receiver=SIC3)
ConnectorControl forwards committed(tid=t2, sender=InventoryService, receiver=SIC3)
ConnectorControl receives commit(tid=t1, sender=SIC1, receiver=InventoryService)
ConnectorControl sends Commit for transaction t1
ConnectorControl receives committed(tid=t1, sender=InventoryService, receiver=SIC1)
ConnectorControl forwards committed(tid=t1, sender=InventoryService, receiver=SIC1)
ConnectorControl receives commit(tid=t5, sender=SIC4, receiver=InventoryService)
ConnectorControl sends Commit for transaction t5
```

RAC sent Abort for t1
RAC sent Commit for t3
RAC sent Ready To Commit for t2
RAC sent Committed for t4
RAC restarted t1
RAC forwarded request held in SPQ

Figure 10.6 Fragment of the execution trace of service RAC for distributed transactions during planned failure scenario

The execution trace (Figure 10.6) shows that after the RAC received a failure notification from the RAM, the RAC received a fifth transaction (t5) which was held in the SPQ. The content of the RAC's queues at this point is as follows:

- The SPQ contains one request that was held due to service failure:
    - PrepareToCommit(t5, SIC4, IS)
- The SAQ contains two prepare to commit requests that have been forwarded to the service:
    - PrepareToCommit (t1, SIC1, IS)
    - PrepareToCommit (t2, SIC3, IS)
- The SRQ contains four requests as follows:
    - PrepareToCommit(t3, SIC2, IS)
    - PrepareToCommit (t4, SIC5, IS)
    - Commit(t3, SIC2, IS)
    - Commit(t4, ISC5, IS)
- The RFQ contains two received responses:
    - ReadyToCommit(t2, IS, SIC3)
    - Committed(t4, IS, SIC5)
- RRQ contains two forwarded responses:
    - ReadyToCommit(t3, IS, SIC2)
    - ReadyToCommit(t4, IS, SIC5)

During the analysis activity which is handled in the Analyzing Failure Events state, the execution trace indicates that the RAC determined a status for each of these active transactions as follows:

- For transaction t1, the RAC determined the status of this transaction as *Preparing*, since the SAQ contains the Prepare To Commit request to the service but neither the RFQ nor the RRQ contain the Ready To Commit response for this transaction.
- For transaction t2, the RAC determined the status of this transaction as *Prepared*, since the SAQ contains the Prepare To Commit request to the service and the RFQ contains the Ready To Commit response for this transaction.
- For transaction t3, the RAC determined the status of this transaction as *Committing*, since the SRQ contains the Commit request for this transaction but neither the RFQ nor the RRQ queues contain the Committed response for this transaction.
- For transaction t4, the RAC determined the status of this transaction as *Committed*, since the SRQ contains the Commit request to the service for this transaction and the RFQ contains a Committed response for this transaction.

During the planning activity which is handled in the Planning for Recovery state, the RAC determined recovery actions for each active transaction as follows:

- For transaction t1, since this transaction failed while being prepared to commit by the service during the first phase of 2PC, the recovery actions determined by the RAC for this transaction after service recovery were (1) to abort this transaction with the service and then (2) to restart this transaction with the recovered service.

- For transaction t2, since this transaction failed after being prepared to commit by the service, the RAC determined a recovery action to send the Ready To Commit for this distributed transaction to the coordinator of this transaction.
- For transaction t3, since the service failed while committing this transaction, the recovery action determined by the RAC for this transaction was to resend the Commit message to the recovered service.
- For transaction t4, the RAC determined a recovery action to send Committed for this distributed transaction to the coordinator of this transaction.

During the execution phase which is handled in the Executing Recovery Plan state, the service RAC recovered the requests of the transactions that must be restarted with the recovered service by moving these requests from the SAQ to the SPQ. In this scenario, only transaction t1 needs to be restarted with the recovered service. Therefore, the execution trace indicates that this message is moved from the SAQ to the SPQ. The content of SPQ after recovery is:

- Prepare(t1, SIC1, IS) // request which was recovered from the SAQ
- Prepare(t5, SIC4, IS) //request held in the SPQ due to service failure

When the RAC is reactivated after the service has recovered, the execution trace indicates that the RAC aborted and then restarted transaction t1 with the recovered service, (2) sent Ready To Commit for transaction t2 to the coordinator of this transaction, (3) requested the recovered service to commit transactions t3, (4) sent Committed for transaction t4 to the coordinator of this distributed transaction, and (5) forwarded transaction t5 which was previously held in the SPQ due to service failure.

The execution trace indicates that service execution resumed normally and that all active transactions were eventually committed.

Self-Configuration Scenario

To illustrate the behavior of the service RAC during adaptation, I use an adaptation scenario that involves adapting the IS. In this scenario, the IS is concurrently processing four distributed transactions (t1-t4). As a result, the expected behavior in this scenario includes:

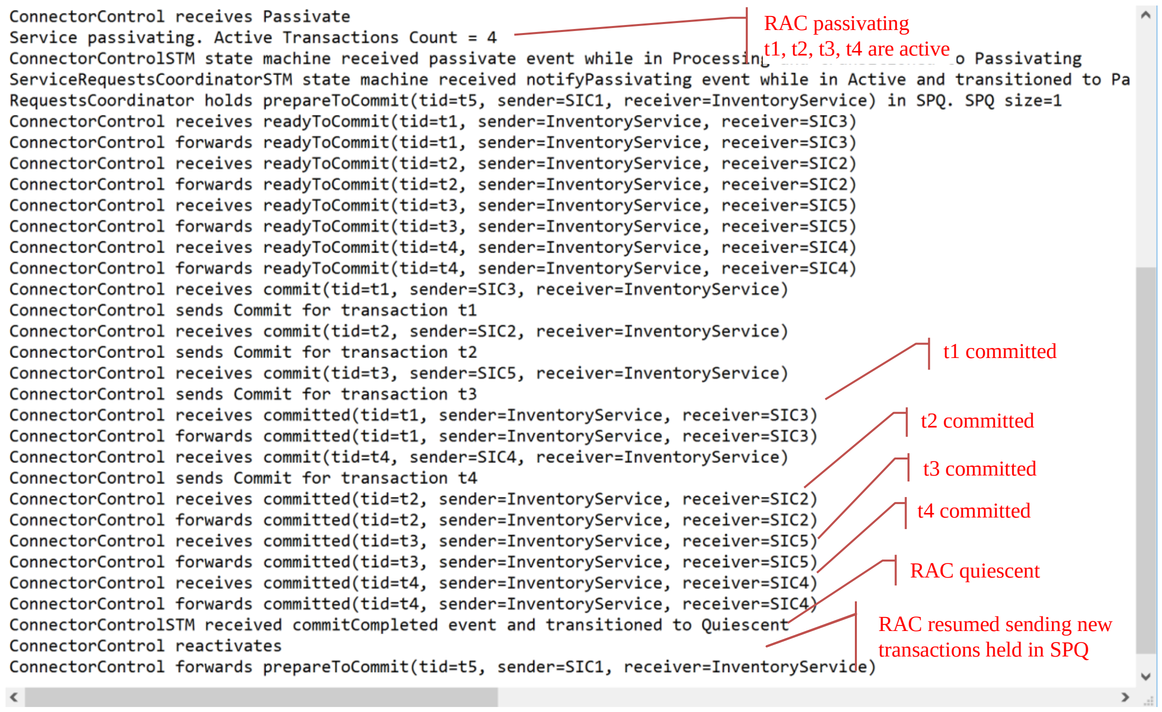
1. When the IS RAC receives the Passivate command, the RAC transitions to the Passivating state in order to allow transactions t1-t4 to complete. Furthermore, the IS RAC holds any new transactions in the SPQ until dynamic adaptation is completed.
2. When transactions t1-t4 complete, the IS RAC transitions to the Quiescent state.
3. When dynamic service adaptation is completed, the IS RAC receives the Reactivate command from the Recovery and Adaptation Manager. As a result, the RAC resumes sending messages to the adapted IS including the transactions held in the SPQ.

When service adaptation is requested, the content of the RAC queues are as follows:

- The SPQ does not contain any request.
- The SAQ contains four prepare to commit requests sent to the service:
    - PrepareToCommit(t1, SIC3, IS)
    - PrepareToCommit (t2, SIC2, IS)
    - PrepareToCommit (t3, SIC5, IS)

    - PrepareToCommit (t4, SIC4, IS)
- The SRQ does not contain any requests.
- The RFQ contains three service responses as follows:
    - ReadyToCommit(t1, IS, SIC3)
    - ReadyToCommit(t2, IS, SIC2)
    - ReadyToCommit(t3, IS, SIC5)
- The RRQ does not indicate any responses that have been forwarded to clients.

As a result of passivation, the execution trace indicates that the service RAC transitioned into the Passivating state (Figure 10.7) where it permitted these four distributed transactions to gradually terminate. While in Passivating state, a new transaction t5 was received and queued by the Service RAC into the SPQ. After the service completed all distributed transactions, the execution trace indicates that the Service RAC transitioned to the Quiescent state at which time the service was dynamically replaced. After adaptation is completed, the service RAC received the reactivate command. As a result, the RAC transitioned to the Active state and forwarded the distributed transaction t5 queued in its SPQ to the service. At this point, normal execution is resumed between RAC and the service.



```
ConnectorControl receives Passivate
Service passivating. Active Transactions Count = 4
ConnectorControlSTM state machine received passivate event while in Processing [illegible]o Passivating
ServiceRequestsCoordinatorSTM state machine received notifyPassivating event while in Active and transitioned to Pa
RequestsCoordinator holds prepareToCommit(tid=t5, sender=SIC1, receiver=InventoryService) in SPQ. SPQ size=1
ConnectorControl receives readyToCommit(tid=t1, sender=InventoryService, receiver=SIC3)
ConnectorControl forwards readyToCommit(tid=t1, sender=InventoryService, receiver=SIC3)
ConnectorControl receives readyToCommit(tid=t2, sender=InventoryService, receiver=SIC2)
ConnectorControl forwards readyToCommit(tid=t2, sender=InventoryService, receiver=SIC2)
ConnectorControl receives readyToCommit(tid=t3, sender=InventoryService, receiver=SIC5)
ConnectorControl forwards readyToCommit(tid=t3, sender=InventoryService, receiver=SIC5)
ConnectorControl receives readyToCommit(tid=t4, sender=InventoryService, receiver=SIC4)
ConnectorControl forwards readyToCommit(tid=t4, sender=InventoryService, receiver=SIC4)
ConnectorControl receives commit(tid=t1, sender=SIC3, receiver=InventoryService)
ConnectorControl sends Commit for transaction t1
ConnectorControl receives commit(tid=t2, sender=SIC2, receiver=InventoryService)
ConnectorControl sends Commit for transaction t2
ConnectorControl receives commit(tid=t3, sender=SIC5, receiver=InventoryService)
ConnectorControl sends Commit for transaction t3
ConnectorControl receives committed(tid=t1, sender=InventoryService, receiver=SIC3)
ConnectorControl forwards committed(tid=t1, sender=InventoryService, receiver=SIC3)
ConnectorControl receives commit(tid=t4, sender=SIC4, receiver=InventoryService)
ConnectorControl sends Commit for transaction t4
ConnectorControl receives committed(tid=t2, sender=InventoryService, receiver=SIC2)
ConnectorControl forwards committed(tid=t2, sender=InventoryService, receiver=SIC2)
ConnectorControl receives committed(tid=t3, sender=InventoryService, receiver=SIC5)
ConnectorControl forwards committed(tid=t3, sender=InventoryService, receiver=SIC5)
ConnectorControl receives committed(tid=t4, sender=InventoryService, receiver=SIC4)
ConnectorControl forwards committed(tid=t4, sender=InventoryService, receiver=SIC4)
ConnectorControlSTM received commitCompleted event and transitioned to Quiescent
ConnectorControl reactivates
ConnectorControl forwards prepareToCommit(tid=t5, sender=SIC1, receiver=InventoryService)
```

Figure 10.7 Fragment of the execution trace of service RAC for distributed transactions during planned adaptation scenario

10.2.4 Experimentation with RAC for Coordinators

Self-Healing Scenario

The coordinator failure scenario demonstrates the ability of the coordinator RAC to recover failed transactions (see section 3.3.4 in chapter 3). In this scenario, the Customer Coordinator (CC) fails while processing one transaction (t1) from Customer Interaction (CI). Therefore, the expected behavior in this scenario includes:

1. The coordinator RAC receives a failure notification from the Recovery and Adaptation Manager.
2. The coordinator RAC determines the appropriate recovery action to recover the failed transaction to send Abort for this transaction and then restarts this transaction with the coordinator when it is recovered.

3. The coordinator RAC receives the Reactivate command from the Recovery and Adaptation Manager to resume sending messages to the recovered CC. As a result, the RAC sends the appropriate recovery actions to the recovered CC as shown in (2).

In this scenario, failure of CC occurred after the CC has initiated the following three sequential transactions:

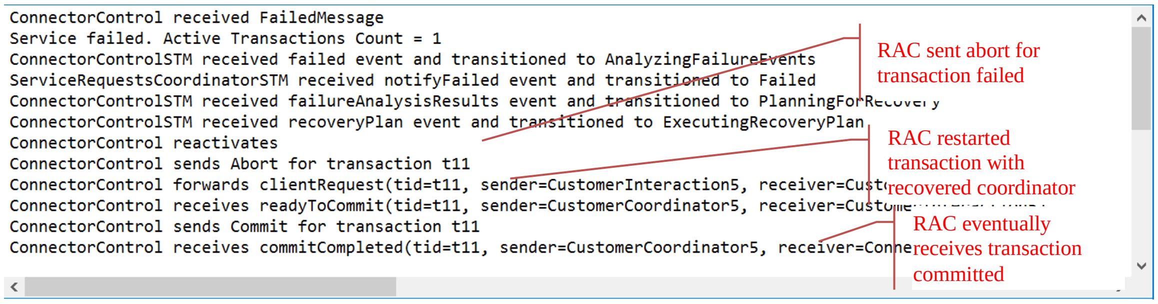
- Before failure, the CC initiated a transaction with the Customer Account Service (CAS) and received the response of this transaction from this service.
- Before failure, the CC initiated a transaction with the Credit Card Service (CCS) and received the response of this transaction from this service.
- Before failure, the CC initiated a transaction with the DOS. However, the CC failed before receiving the response of this transaction from this service.

In this recovery scenario, the execution trace (Figure 10.8) indicates that when the CC has recovered, the coordinator RAC (1) instructed the recovered CC to abort the CI transaction and then (2) restarted this transaction with the recovered CC. Furthermore, the execution traces of the CAS RAC, CCS RAC, and DOS RAC indicate that these RACs received duplicate requests from the recovered CC since the recovered coordinator restarted the same transactions with these services. The RAC of each of these services reacted to these duplicate requests as follows:

- The execution trace of the CAS RAC indicates that this RAC discarded the duplicate request and sent back to the recovered CC the response of this request using its Response Recovery Queue (RRQ).

- The execution trace of the CCS RAC indicates that this RAC discarded this duplicate request and sent back to the recovered CC the response of this request using its Response Recovery Queue (RRQ).
- The execution trace of the DOS RAC indicates that this RAC discarded the duplicate request. The execution trace also showed that this RAC has not yet received the response of the original request from the DOS (i.e., DOS is still processing the original request). Therefore, when this RAC received the response of this request from DOS, it forwarded that response to the recovered CC.

After receiving the response from the DOS RAC, the execution trace showed that the recovered CC continued working on this transaction until it has completed.



```
ConnectorControl received FailedMessage
Service failed. Active Transactions Count = 1
ConnectorControlSTM received failed event and transitioned to AnalyzingFailureEvents
ServiceRequestsCoordinatorSTM received notifyFailed event and transitioned to Failed
ConnectorControlSTM received failureAnalysisResults event and transitioned to PlanningForRecovery
ConnectorControlSTM received recoveryPlan event and transitioned to ExecutingRecoveryPlan
ConnectorControl reactivates
ConnectorControl sends Abort for transaction t11
ConnectorControl forwards clientRequest(tid=t11, sender=CustomerInteraction5, receiver=Cust
ConnectorControl receives readyToCommit(tid=t11, sender=CustomerCoordinator5, receiver=Custome
ConnectorControl sends Commit for transaction t11
ConnectorControl receives commitCompleted(tid=t11, sender=CustomerCoordinator5, receiver=Conne
```

RAC sent abort for transaction failed
RAC restarted transaction with recovered coordinator
RAC eventually receives transaction committed

Figure 10.8 Fragment of the execution trace of coordinator RAC during planned failure scenario

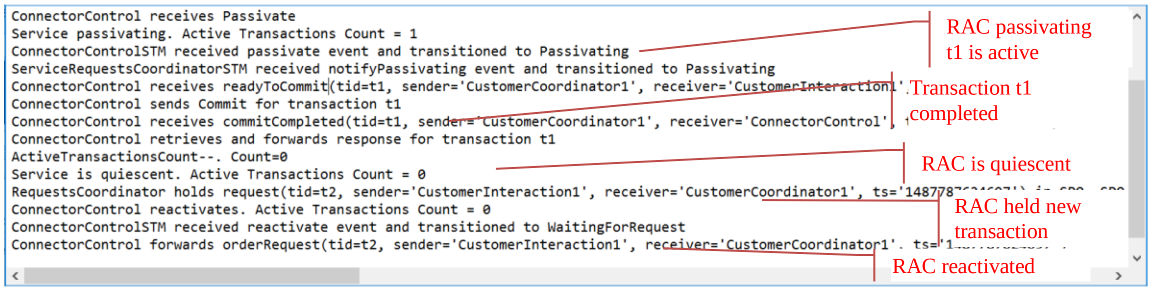
Self-Configuration Scenario

The coordinator adaptation scenario is an experiment to validate the adaptation of the coordinator as described in chapter 3. In this scenario, passivation of CC is requested while the CC is preparing to commit a transaction. As a result, the expected behavior in this scenario includes:

1. When the CC RAC receives the Passivate command, the RAC transitions to the Passivating state in order to allow the active transaction to complete.
2. When this transaction completes, the CC RAC transition to the Quiescent state. Furthermore, the CC RAC holds any new transactions in the SPQ until dynamic adaptation is completed.
3. When the CC RAC receives the Reactivate command from the Recovery and Adaptation Manager, it resumes sending messages to the adapted CC including the transaction held in the SPQ.

The execution trace (Figure 10.9) of the CC RAC shows that when the RAC received the Passivate command, the CC RAC transitioned to Passivating state. When the transaction ended with the sending of the coordinator response to the client, the state machine transitioned to the Quiescent state. While the CC RAC is in quiescent state, it received and queued a new transaction from the CI in the Coordinator Pending Queue.

After replacing CC, the CC RAC received the reactivate message from the external RAM and then transitioned from the Quiescent to the Waiting for Request state. As a result, the RAC forwarded the queued transaction t2 to CC.



```
ConnectorControl receives Passivate
Service passivating. Active Transactions Count = 1
ConnectorControlSTM received passivate event and transitioned to Passivating
ServiceRequestsCoordinatorSTM received notifyPassivating event and transitioned to Passivating
ConnectorControl receives readyToCommit(tid=t1, sender='CustomerCoordinator1', receiver='CustomerInteraction1'
ConnectorControl sends Commit for transaction t1
ConnectorControl receives commitCompleted(tid=t1, sender='CustomerCoordinator1', receiver='ConnectorControl',
ConnectorControl retrieves and forwards response for transaction t1
ActiveTransactionsCount--. Count=0
Service is quiescent. Active Transactions Count = 0
RequestsCoordinator holds request(tid=t2, sender='CustomerInteraction1', receiver='CustomerCoordinator1', ts='148778
ConnectorControl reactivates. Active Transactions Count = 0
ConnectorControlSTM received reactivate event and transitioned to WaitingForRequest
ConnectorControl forwards orderRequest(tid=t2, sender='CustomerInteraction1', receiver='CustomerCoordinator1', ts='1
```

Figure 10.9 Fragment of the execution trace of coordinator RAC during planned adaptation scenario

10.3 Experimentation with RAC in asynchronous patterns

This section describes the experimental design and results of the consumer RAC used in asynchronous patterns described in chapter 4. In order to carry out experimentation, I implemented the architecture of the Emergency Monitoring System (EMS) case study (Gomaa, 2011) (Figure 10.10) in which an operator can view various alarm events generated by sensor components and also can request the status of these sensors. This case study is chosen to experiment with the consumer RAC since it consists of several asynchronous patterns including the unidirectional asynchronous message communication and the subscription/notification patterns.

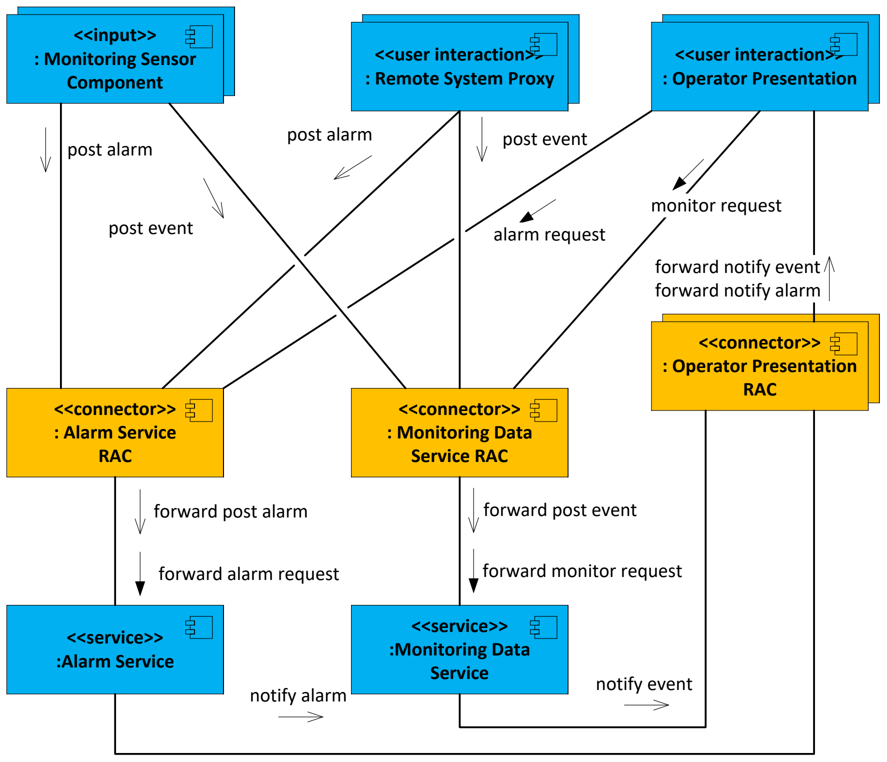
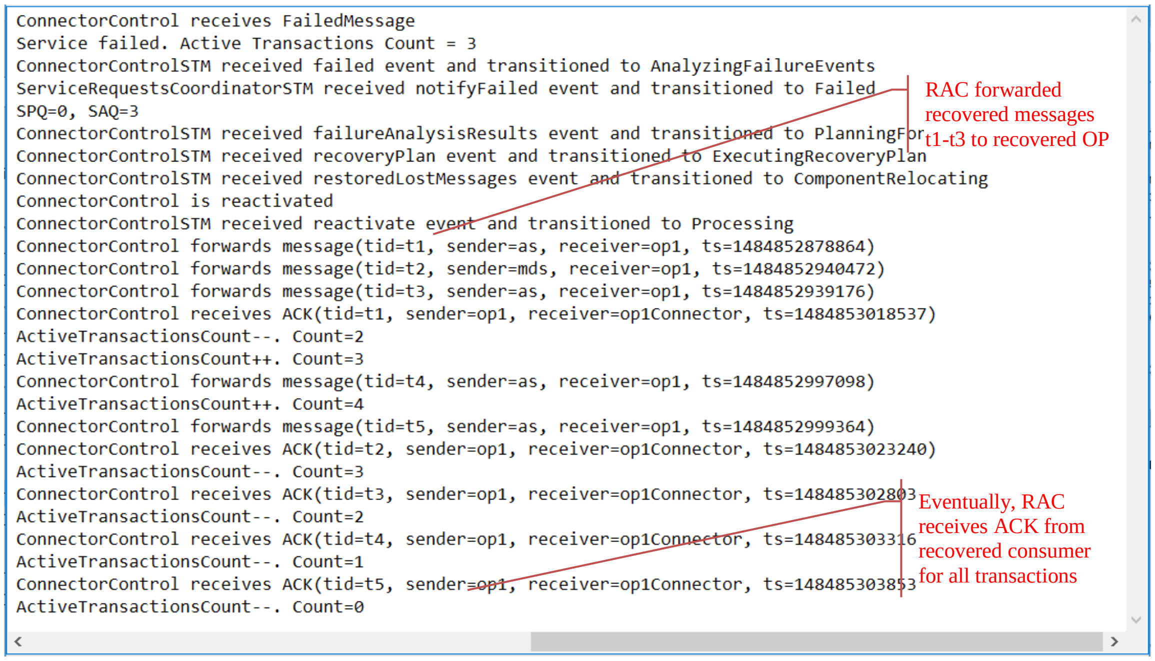


Figure 10.10 The Emergency Monitoring System (EMS) architecture

Self-Healing Scenario

The consumer failure scenario demonstrates the ability of the consumer RAC to recover failed transactions. In this scenario, the Operator Presentation (OP) component is concurrently processing three active transactions (t1-t3) at the time of failure. As described previously in chapter 4, an asynchronous transaction is considered failed if (1) the consumer RAC forwarded an asynchronous message to the consumer and (2) the consumer RAC did not receive a corresponding ACK message for this transaction from the consumer. Therefore, the OP RAC in this scenario must recover all failed transactions with the OP consumer when it is recovered. The expected behavior in this scenario includes:

1. The OP RAC receives a failure notification from the Recovery and Adaptation Manager.
2. The OP RAC determines recovery actions to recover the failed transactions t1-t3.
3. The OP RAC receives the Reactivate command from the Recovery and Adaptation Manager to resume sending messages to the recovered OP. As a result, OP RAC sends transactions t1-t3 to the recovered OP.



```
ConnectorControl receives FailedMessage
Service failed. Active Transactions Count = 3
ConnectorControlSTM received failed event and transitioned to AnalyzingFailureEvents
ServiceRequestsCoordinatorSTM received notifyFailed event and transitioned to Failed
SPQ=0, SAQ=3
ConnectorControlSTM received failureAnalysisResults event and transitioned to PlanningFor
ConnectorControlSTM received recoveryPlan event and transitioned to ExecutingRecoveryPlan
ConnectorControlSTM received restoredLostMessages event and transitioned to ComponentRelocating
ConnectorControl is reactivated
ConnectorControlSTM received reactivate event and transitioned to Processing
ConnectorControl forwards message(tid=t1, sender=as, receiver=op1, ts=1484852878864)
ConnectorControl forwards message(tid=t2, sender=mds, receiver=op1, ts=1484852940472)
ConnectorControl forwards message(tid=t3, sender=as, receiver=op1, ts=1484852939176)
ConnectorControl receives ACK(tid=t1, sender=op1, receiver=op1Connector, ts=1484853018537)
ActiveTransactionsCount--. Count=2
ActiveTransactionsCount++. Count=3
ConnectorControl forwards message(tid=t4, sender=as, receiver=op1, ts=1484852997098)
ActiveTransactionsCount++. Count=4
ConnectorControl forwards message(tid=t5, sender=as, receiver=op1, ts=1484852999364)
ConnectorControl receives ACK(tid=t2, sender=op1, receiver=op1Connector, ts=1484853023240)
ActiveTransactionsCount--. Count=3
ConnectorControl receives ACK(tid=t3, sender=op1, receiver=op1Connector, ts=148485302803
ActiveTransactionsCount--. Count=2
ConnectorControl receives ACK(tid=t4, sender=op1, receiver=op1Connector, ts=148485303316
ActiveTransactionsCount--. Count=1
ConnectorControl receives ACK(tid=t5, sender=op1, receiver=op1Connector, ts=148485303853
ActiveTransactionsCount--. Count=0
```

RAC forwarded recovered messages t1-t3 to recovered OP
Eventually, RAC receives ACK from recovered consumer for all transactions

Figure 10.11 Fragment of the execution trace of the consumer RAC during planned failure scenario

In this scenario, the execution trace (Figure 10.11) indicates that the content of the RAC's queues at the time of failure is as follows:

- The SPQ does not contain any pending messages to the OP component.

- The SAQ contains three messages that have been forwarded to the OP as follows:
    - Message(t1, AS, OP1)
    - Message(t2, MDS, OP1)
    - Message(t3, AS, OP1)

During the analysis activity which is handled in the Analyzing Failure Events state, the execution trace indicates that the OP RAC determined that transactions *t1-t3* had failed, since these asynchronous transactions have been forwarded by the OP RAC to the OP consumer but consumer failure occurred before the OP RAC received the corresponding ACK messages for these transactions. As a result, the execution trace indicates that the OP RAC moved these transactions from the SAQ to the SPQ. The execution trace indicates that the states of the SPQ and SAQ after recovery are as follows:

- The SPQ contains three recovered messages as follows:
    - Message(t1, AS, OP1)
    - Message(t2, MDS, OP1)
    - Message(t3, AS, OP1)
- The SAQ is empty.

When the RAC is reactivated after the OP has recovered, the execution trace indicates that the RAC sent lost transactions t1-t3 to the recovered OP consumer and that the recovered OP sent corresponding ACK messages for these transactions to the RAC indicating that it is done with these transactions. At this point consumer execution resumed normally.

Self-Configuration Scenario

To illustrate the behavior of the consumer RAC during adaptation (see section 4.1 in chapter 4) , I use an adaptation scenario that involves adapting the OP component. In this scenario, the OP is concurrently processing four transactions (t1-t4). As a result, the expected behavior in this scenario includes:

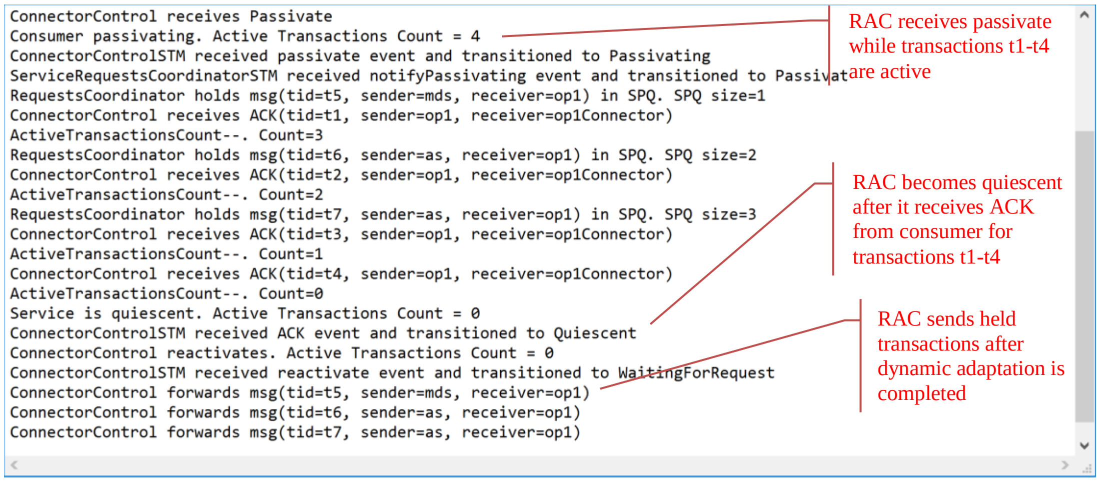
1. When the OP RAC receives the Passivate command, the RAC transitions to the Passivating state until it receives ACK messages from the OP for transactions t1-t4. Furthermore, the OP RAC holds any new transactions to the OP in the SPQ until dynamic adaptation is completed.
2. When the OP RAC receives ACKs for transactions t1-t4, the RAC transitions to the Quiescent state in which the OP can be safely adapted.
3. The OP RAC receives a Reactivate command from the Recovery and Adaptation Manager so that the RAC resumes sending messages to the adapted OP. As a result, OP RAC sends any held transactions in the SPQ to the adapted OP.

In this scenario, the contents of the various queues when the consumer RAC received the Passivate command are as follows:

- The SPQ does not contain any pending messages.
- The SAQ contains four active messages that have been forwarded to the consumer:
    - Message(t1, MDS, OP1)
    - Message(t2, AS, OP1)
    - Message(t3, MDS, OP1)

- Message(t4, AS, OP1)

As a result of passivation, the execution trace indicates that the OP RAC transitioned into the Passivating state (Figure 10.12) until the consumer sends ACK messages to the RAC indicating that it is done with these transactions. While in the passivating state, the OP RAC received new transactions. As a result, the RAC queued these transactions into the Pending Queue. After the OP has acknowledged all four active transactions to the OP RAC, the execution trace indicates that the OP RAC transitioned to the Quiescent state at which time the OP was dynamically replaced. After adaptation is completed, the OP RAC received the reactivate command. At this point, normal execution is resumed between RAC and the consumer. The execution trace indicates that the OP RAC forwarded all transactions held previously in the pending queue due to dynamic adaptation to the OP.



```
ConnectorControl receives Passivate
Consumer passivating. Active Transactions Count = 4
ConnectorControlSTM received passivate event and transitioned to Passivating
ServiceRequestsCoordinatorSTM received notifyPassivating event and transitioned to Passivat
RequestsCoordinator holds msg(tid=t5, sender=mds, receiver=op1) in SPQ. SPQ size=1
ConnectorControl receives ACK(tid=t1, sender=op1, receiver=op1Connector)
ActiveTransactionsCount--. Count=3
RequestsCoordinator holds msg(tid=t6, sender=as, receiver=op1) in SPQ. SPQ size=2
ConnectorControl receives ACK(tid=t2, sender=op1, receiver=op1Connector)
ActiveTransactionsCount--. Count=2
RequestsCoordinator holds msg(tid=t7, sender=as, receiver=op1) in SPQ. SPQ size=3
ConnectorControl receives ACK(tid=t3, sender=op1, receiver=op1Connector)
ActiveTransactionsCount--. Count=1
ConnectorControl receives ACK(tid=t4, sender=op1, receiver=op1Connector)
ActiveTransactionsCount--. Count=0
Service is quiescent. Active Transactions Count = 0
ConnectorControlSTM received ACK event and transitioned to Quiescent
ConnectorControl reactivates. Active Transactions Count = 0
ConnectorControlSTM received reactivate event and transitioned to WaitingForRequest
ConnectorControl forwards msg(tid=t5, sender=mds, receiver=op1)
ConnectorControl forwards msg(tid=t6, sender=as, receiver=op1)
ConnectorControl forwards msg(tid=t7, sender=as, receiver=op1)
```

RAC receives passivate while transactions t1-t4 are active
RAC becomes quiescent after it receives ACK from consumer for transactions t1-t4
RAC sends held transactions after dynamic adaptation is completed

Figure 10.12 Fragment of the execution trace of the consumer RAC during planned adaptation scenario

10.4 Experimental Analysis of the DARE framework

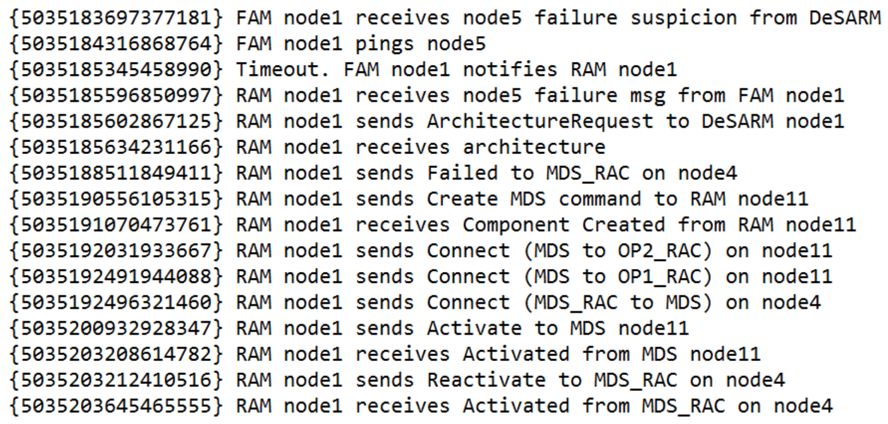
Chapter 5 described the design of the DARE framework and showed how it can handle failure recovery and dynamic adaptation of components in CBSAs. In order to validate the design of the DARE framework, I conducted two types of experiments using the Emergency Monitoring System (see Figure 10.10) case study: a self-healing scenario and a self-configuration scenario. The self-healing scenario illustrates DARE's capability of recovering from node failures. This scenario consists of taking down the node that hosts the Monitoring Data Service (MDS) and then inspecting the execution trace to determine that DARE was able to detect and dynamically recover this component on a different node. The self-configuration scenario illustrates DARE's capability of dynamically adapting the EMS architecture.

10.4.1 Self-Healing Scenario

In the self-healing scenario, node 5, which hosts the MDS component, has failed. As a result, the expected behavior of DARE during this experiment include (1) DeSARM detects failure of node 5, (2) FAM pings node 5 to confirm failure of node 5 and then activates the RAM to recover components hosted by node 5, (3) the Recovery and Adaptation Manager (RAM) plans and executes the recovery actions to recover the failed MDS component by notifying the MDS RAC of failure, recovering another instance of the MDS on a different node, and then activating the recovered MDS and the MDS RAC.

A fragment of the execution trace illustrating major events of this scenario is shown in Figure 10.13. The trace indicates that DeSARM (Figure 10.14) on every healthy node suspected the failure of node 5 due to absence of gossip messages from that node. As a result, each DeSARM sent a node 5 failure notification to the peer FAM

(Figure 10.14). Although FAMs on multiple nodes have been activated, the execution trace indicates that only the FAM hosted by the node with the lowest IP address (node 1 in Figure 10.13), and thus the recovery node, proceeded with the recovery process by pinging node 5. Since node 5 failed, the FAM on node 1 did not receive a heartbeat reply from this failed node. As a result, the FAM on node 1 notified the RAM on node 1 of the failed node.



```
{5035183697377181} FAM node1 receives node5 failure suspicion from DeSARM
{5035184316868764} FAM node1 pings node5
{5035185345458990} Timeout. FAM node1 notifies RAM node1
{5035185596850997} RAM node1 receives node5 failure msg from FAM node1
{5035185602867125} RAM node1 sends ArchitectureRequest to DeSARM node1
{5035185634231166} RAM node1 receives architecture
{5035188511849411} RAM node1 sends Failed to MDS_RAC on node4
{5035190556105315} RAM node1 sends Create MDS command to RAM node11
{5035191070473761} RAM node1 receives Component Created from RAM node11
{5035192031933667} RAM node1 sends Connect (MDS to OP2_RAC) on node11
{5035192491944088} RAM node1 sends Connect (MDS to OP1_RAC) on node11
{5035192496321460} RAM node1 sends Connect (MDS_RAC to MDS) on node4
{5035200932928347} RAM node1 sends Activate to MDS node11
{5035203208614782} RAM node1 receives Activated from MDS node11
{5035203212410516} RAM node1 sends Reactivate to MDS_RAC on node4
{5035203645465555} RAM node1 receives Activated from MDS_RAC on node4
```

Figure 10.13 Fragment of execution trace during the self-healing scenario

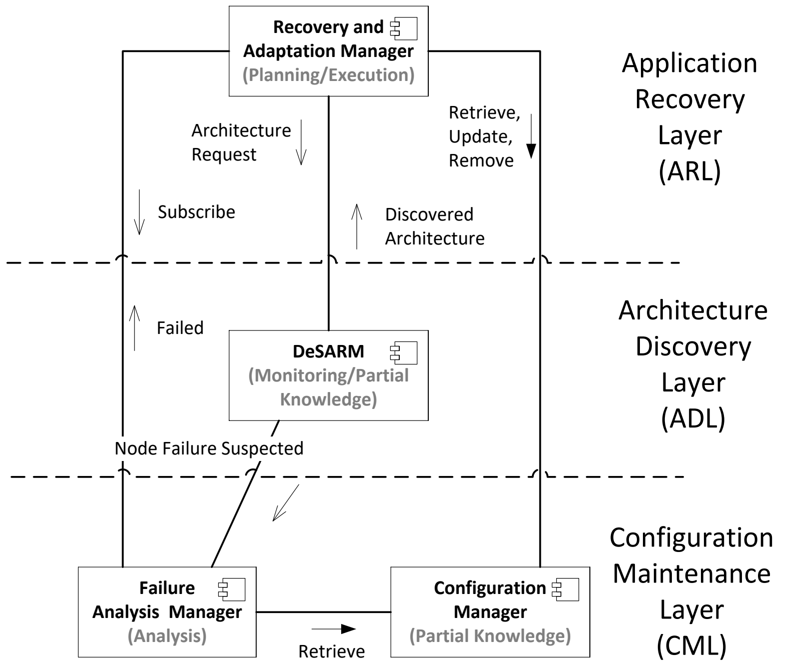


Figure 10.14 The DARE architecture

The execution trace indicates that the RAM on node 1 first requested the architecture from DeSARM and then proceeded with the recovery process by retrieving from the Configuration Manager the set of identifiers of components hosted by the failed node 5. The execution trace indicates that node 5 hosted only one component, namely the MDS. Inspection of the detailed execution trace also indicates that the RAM proceeded to determine the recovery plan as follows:

- From the architecture obtained from DeSARM and from the configuration map, the RAM determined that node 4 hosts the input RAC of the MDS component because this RAC forwards synchronous requests and asynchronous messages to the failed MDS.

- The RAM on node 1 notified the MDS RAC on node 4 of component failure so that this RAC ceases forwarding messages to the MDS and starts recovering any failed transactions.
- From the DeSARM architecture, the RAM determined the recipient RACs that receive messages from the failed MDS, which in this scenario were two instances of the Operator Presentation (OP) RAC. These recipient RACs are determined by the RAM so that the recovered MDS can be connected with these RACs.
- The RAM selected node 11 to host the recovered MDS.

When planning is complete, the execution trace indicates that the RAM on node 1 requested the RAM on node 11 to create the MDS component. When the component is created, the execution trace indicates that the RAM on node 1 requested the MDS RAC to connect to the recovered MDS and requested the recovered MDS to connect to the OP RACs. Finally, the execution trace indicates that the RAM on node 1 activated the recovered MDS and then reactivated the MDS RAC. As a result, the MDS RAC resumed sending messages to the recovered MDS, including any lost messages due to failure.

In order to assess the effect of DARE's decentralization on recovery time, I measured the average recovery time that the DARE framework takes to recover the MDS component, starting from the time that DeSARM at the recovery node sent a notification message to the FAM indicating that a node failure is suspected to the time that the MDS RAC is reactivated indicating that normal communication with the recovered MDS is resumed. In order to do this, I ran 5 experiments sets such that in each set I varied the number of nodes in the system. I experimented with recovery when the number of nodes is 11, 16, 21, 26, and 30 nodes, with the corresponding increase in component instances and RACs. Each experiment consisted of one backup node for hosting the recovered component.

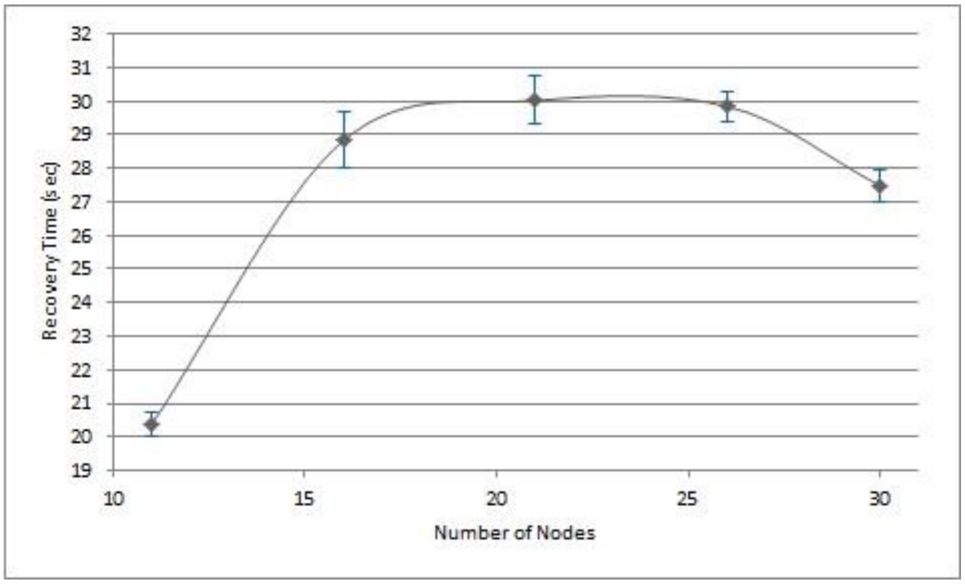


Figure 10.15 Average recovery time during the self-healing scenario

In each experiment set, I ran the self-healing scenario 30 times. The results in Figure 10.15 show the 99% confidence intervals of the measured recovery times for the 5 experiment sets. When the system size is 11 nodes, the average recovery time is 20.4 seconds. Doubling the system size from 11 nodes to 21 nodes caused the average recovery time to increase by approximately 47%. However, increasing the system size from 21 to 30 nodes caused the recovery time to drop slightly by 8%, which shows

DARE's capability to scale up as the system size increases. These results can be attributed to the use of a Distributed Hash Table (DHT) to store the distributed configuration map since DHTs scale logarithmically (Stoica et al., 2003) with the system size.

10.4.2 Self-Configuration Scenario

In the self-configuration scenario, the RAM on node 1 received an external adaptation request to adapt the Alarm Service (AS). In this scenario, I applied the load balancing pattern to the Alarm Service (AS), as shown in Figure 10.16. This pattern involves replacing the AS with a load balancer component and two instances, AS1 and AS2, of the AS. The load balancer is responsible for forwarding messages to the AS instances in a simple round-robin fashion.

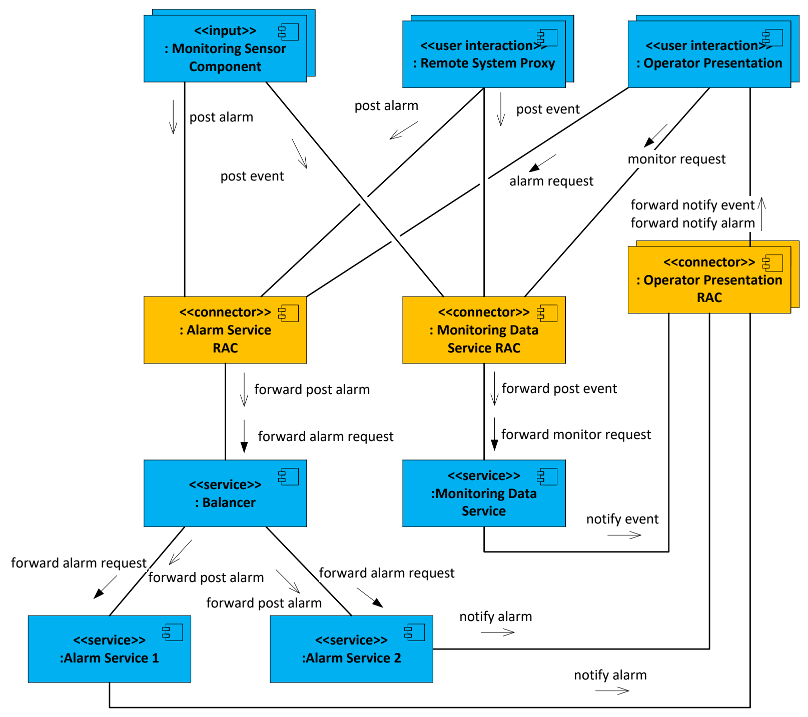
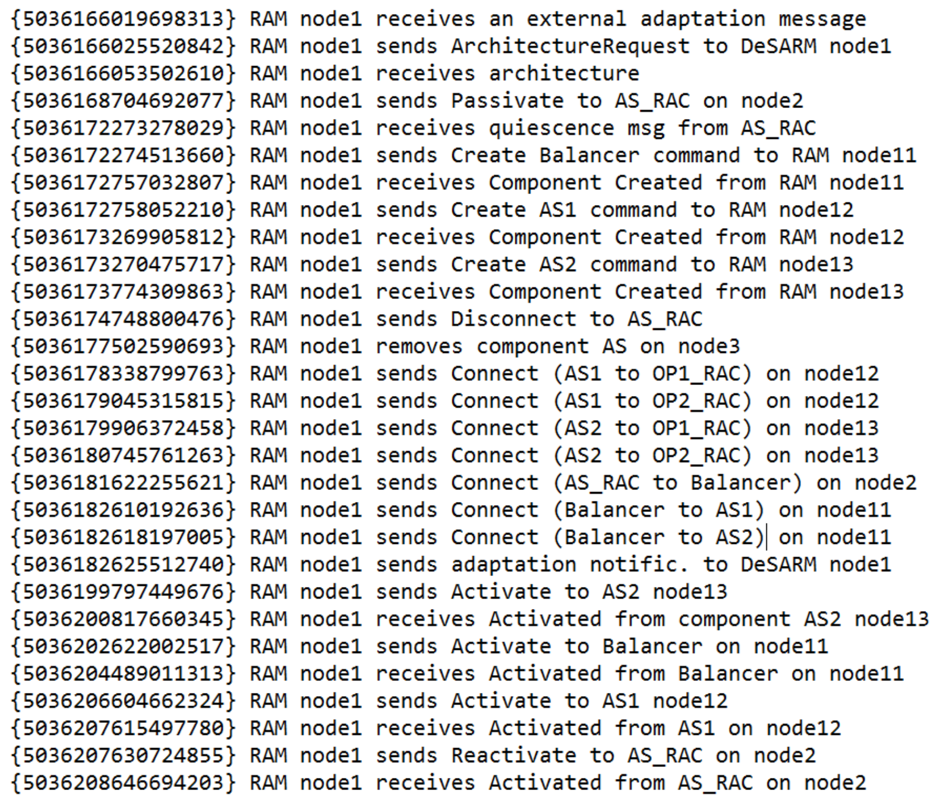


Figure 10.16 EMS architecture after dynamic adaptation

The expected behavior of DARE during this experiment include (1) the RAM sends the Passivate command to the AS RAC so that the RAC steers the AS to the quiescent state, (2) the RAM create the load balancer component, and two instances of AS (AS1 and AS2) (3) the RAM disconnects and removes the previous AS from the architecture, (4) the RAM connects components affected by adaptation (5) the RAM updates the configuration and requests DeSARM to initiate discovery of the adapted architecture, and finally (6) the RAM activates the new components and the AS RAC.

The execution trace (Figure 10.17) shows the major events executed during this scenario. After receiving the external adaptation request, the RAM requested the current software architecture (see chapter 5) from DeSARM (Porter et al., 2016), also located on node 1. The RAM then used the architecture to determine the input RAC of the AS (i.e., the Alarm Service RAC in Figure 10.10). As a result, the detailed execution trace indicates that the RAM on node 1 retrieved the location of the AS RAC from the CM and then sent the Passivate command to the AS RAC hosted by node 2. As a result, the AS RAC steered the AS to a quiescent state, held any input messages to the AS in its queues until dynamic adaptation was complete, and then notified the RAM on node 1 of AS quiescence. The execution trace also indicates that the RAM planned to create the load balancer on node 11, the AS1 on node 12, and the AS2 on node 13.

The RAM on node 1 then proceeded with the dynamic adaptation process by (1) requesting the RAMs on nodes 11, 12, and 13 to create the load balancer component, the AS1, and the AS2, respectively (2) disconnecting and removing the AS from the architecture, (3) connecting the AS1 with the OP1 RAC and the OP2 RAC, (4) connecting the AS2 with the OP1 RAC and the OP2 RAC, (5) connecting the AS RAC with the load balancer, (6) connecting the load balancer with the AS1 and AS2, (7) updating the configuration and requesting DeSARM to initiate discovery of the adapted architecture, and finally (5) activating the new components on nodes 11-13 and the AS RAC on node 2.



```
{5036166019698313} RAM node1 receives an external adaptation message
{5036166025520842} RAM node1 sends ArchitectureRequest to DeSARM node1
{5036166053502610} RAM node1 receives architecture
{5036168704692077} RAM node1 sends Passivate to AS_RAC on node2
{5036172273278029} RAM node1 receives quiescence msg from AS_RAC
{5036172274513660} RAM node1 sends Create Balancer command to RAM node11
{5036172757032807} RAM node1 receives Component Created from RAM node11
{5036172758052210} RAM node1 sends Create AS1 command to RAM node12
{5036173269905812} RAM node1 receives Component Created from RAM node12
{5036173270475717} RAM node1 sends Create AS2 command to RAM node13
{5036173774309863} RAM node1 receives Component Created from RAM node13
{5036174748800476} RAM node1 sends Disconnect to AS_RAC
{5036177502590693} RAM node1 removes component AS on node3
{5036178338799763} RAM node1 sends Connect (AS1 to OP1_RAC) on node12
{5036179045315815} RAM node1 sends Connect (AS1 to OP2_RAC) on node12
{5036179906372458} RAM node1 sends Connect (AS2 to OP1_RAC) on node13
{5036180745761263} RAM node1 sends Connect (AS2 to OP2_RAC) on node13
{5036181622255621} RAM node1 sends Connect (AS_RAC to Balancer) on node2
{5036182610192636} RAM node1 sends Connect (Balancer to AS1) on node11
{5036182618197005} RAM node1 sends Connect (Balancer to AS2) on node11
{5036182625512740} RAM node1 sends adaptation notific. to DeSARM node1
{5036199797449676} RAM node1 sends Activate to AS2 node13
{5036200817660345} RAM node1 receives Activated from component AS2 node13
{5036202622002517} RAM node1 sends Activate to Balancer on node11
{5036204489011313} RAM node1 receives Activated from Balancer on node11
{5036206604662324} RAM node1 sends Activate to AS1 node12
{5036207615497780} RAM node1 receives Activated from AS1 on node12
{5036207630724855} RAM node1 sends Reactivate to AS_RAC on node2
{5036208646694203} RAM node1 receives Activated from AS_RAC on node2
```

Figure 10.17 Fragment of execution trace during the self-configuration scenario

10.5 Experimentation with the Assistant RAC

This section describes the validation of the Assistant RAC (ARAC) which handles adaptation and recovery concerns of senders (see chapter 6). This experiment involves adapting and recovering the Customer Interaction component in the Online Shopping System (see Figure 10.1 in section 10.2).

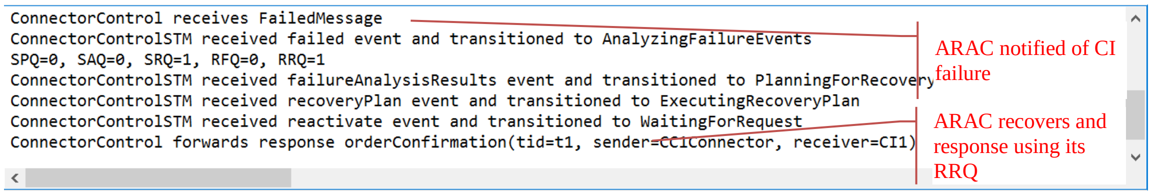
Self-Healing Scenario

The sender failure scenario demonstrates the ability of the Assistant RAC (ARAC) to recover failed responses to the sender. In this scenario, the Customer Interaction (CI) failed after it has initiated a transaction with the Customer Coordinator (CC). As described previously in chapter 6, when the CI recovers from failure, the CI

ARAC must resend the last response in its response queues to the recovered CI.

Therefore, the expected behavior of the CI ARAC in this scenario includes:

1. The CI ARAC receives a failure notification from the Recovery and Adaptation Manager indicating failure of the CI.
2. The CI ARAC determines the recovery action to recover the last response it received from the coordinator.
3. The CI ARAC receives the Reactivate command from the Recovery and Adaptation Manager. As a result, the CI RAC sends the last response to the recovered CI.



```
ConnectorControl receives FailedMessage
ConnectorControlSTM received failed event and transitioned to AnalyzingFailureEvents
SPQ=0, SAQ=0, SRQ=1, RFQ=0, RRQ=1
ConnectorControlSTM received failureAnalysisResults event and transitioned to PlanningForRecovery
ConnectorControlSTM received recoveryPlan event and transitioned to ExecutingRecoveryPlan
ConnectorControlSTM received reactivate event and transitioned to WaitingForRequest
ConnectorControl forwards response orderConfirmation(tid=t1, sender=CC1Connector, receiver=CI1)
```

ARAC notified of CI failure
ARAC recovers and response using its RRQ

Figure 10.18 Fragment of the execution trace of the ARAC during planned failure scenario

In this scenario, the execution trace (Figure 10.18) indicates that the content of the ARAC's queues at the time of failure is as follows:

- The SPQ and the SAQ do not contain any pending or active requests.
- The SRQ contains one request as follows:
    - request(t1, CC1, CC1Connector)
- The RFQ does not contain any responses.
- The RRQ contains one forwarded response:

    - response(t1, CC1Connector, CI1)

During the analysis activity which is handled in the Analyzing Failure Events state, the execution trace indicates that the CI ARAC determined that the response in its RRQ for transaction t1 must be recovered and sent to the CI1 when it recovers. Therefore, CI ARAC proceeded with recovery by moving this response from the RRQ to the RFQ. The content of the RRQ after recovery is as follows:

- The RFQ contains one recovered response as follows:
    - response(t1, CC1Connector, CI1)

When the ARAC is reactivated after the CI has recovered, the execution trace indicates that the ARAC sent the recovered response to the recovered CI, as expected for this scenario.

Self-Configuration Scenario

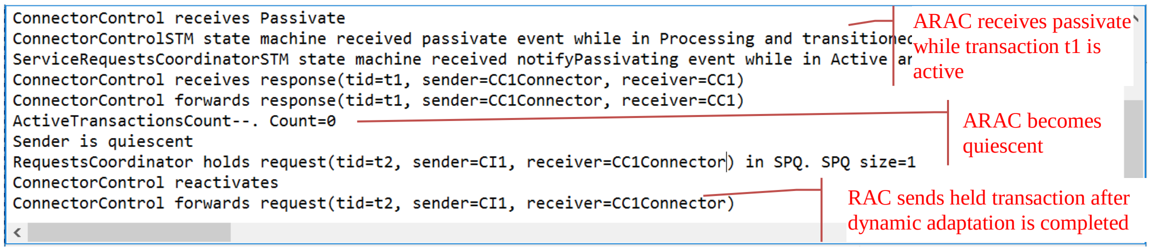
To illustrate the behavior of the ARAC during adaptation, I use an adaptation scenario that involves adapting the CI component. In this scenario, the CI initiated one transaction (t1) with its coordinator. As a result, the expected behavior in this scenario includes:

1. When the CI ARAC receives the Passivate command, the ARAC transitions to the Passivating state until t1 is completed.
2. When t1 is completed, the CI ARAC transitions to the Quiescent state. Furthermore, the CI ARAC holds any new transactions until dynamic adaptation is completed.
3. The CI ARAC receives a Reactivate command from the Recovery and Adaptation Manager. As a result, CI ARAC resumes sending held transactions in the SPQ.

In this scenario, the contents of the various queues when the ARAC received the Passivate command are as follows:

- The SPQ does not contain any transactions.
- The SAQ contains one active request:
    - Request2(t1, CI1, CC1Connector)
- The SRQ contains one request:
    - Request1(t1, CI1, CC1Connector)
- The RFQ does not contain any responses.
- The RRQ contains one response:
    - Response1(t1, CC1Connector, CI1)

As a result of passivation, the execution trace indicates that the ARAC transitioned into the Passivating state (Figure 10.19) until transaction t1 has completed. When t1 is completed, the ARAC transitioned to the Quiescent state. While in the Quiescent state, the ARAC received and held a new transaction t2 in the SPQ. After adaptation is completed, the ARAC received the reactivate command. At this point, normal execution is resumed between ARAC and the CI. The execution trace indicates that when adaptation was completed, the ARAC forwarded transaction t2, which was previously held in the pending queue, to the CC1 Connector.



```
ConnectorControl receives Passivate
ConnectorControlSTM state machine received passive event while in Processing and transitionec
ServiceRequestsCoordinatorSTM state machine received notifyPassivating event while in Active ar
ConnectorControl receives response(tid=t1, sender=CC1Connector, receiver=CC1)
ConnectorControl forwards response(tid=t1, sender=CC1Connector, receiver=CC1)
ActiveTransactionsCount--. Count=0
Sender is quiescent
RequestsCoordinator holds request(tid=t2, sender=CI1, receiver=CC1Connector) in SPQ. SPQ size=1
ConnectorControl reactivates
ConnectorControl forwards request(tid=t2, sender=CI1, receiver=CC1Connector)
```

ARAC receives passive while transaction t1 is active
ARAC becomes quiescent
RAC sends held transaction after dynamic adaptation is completed

Figure 10.19 Fragment of the execution trace of the ARAC during planned adaptation scenario

10.6 Experimentation with RAC Recovery

This section describes recovery of the RAC after a run-time failure occurring during normal execution. In this experiment, the Customer Coordinator RAC fails after forwarding a transaction t1 to the Customer Coordinator (CC). At the time of RAC failure, t1 is a non-distributed transaction that is in the Preparing To Commit state, since the RAC sent the Prepare To Commit message to the CC but the RAC failed before receiving Ready To Commit from the CC. The expected behavior in this experiment is as follows:

1. The RAM recovers another instance of CC RAC.
2. The recovered CC RAC reconstructs its state by replaying messages from its log.
3. The recovered CC RAC recovers transaction t1 by instructing CC to abort this transaction and then restart this transaction with the CC.

The execution trace (Figure 10.20) indicates that after the RAC recovered its state by replaying messages from its log, the SAQ queues one Prepare To Commit message for transaction t1, which corresponds to the same state that RAC had before failure. Furthermore, based on the content of the queues, the RAC determined that transaction t1 must be aborted and then restarted with the CC, since this transaction is in the Preparing to Commit state. As a result, the recovered RAC instructed the CC to abort transaction t1and then restarted this transaction with CC. The execution trace indicates that transaction t1 eventually completed, which is the expected outcome for this experiment.

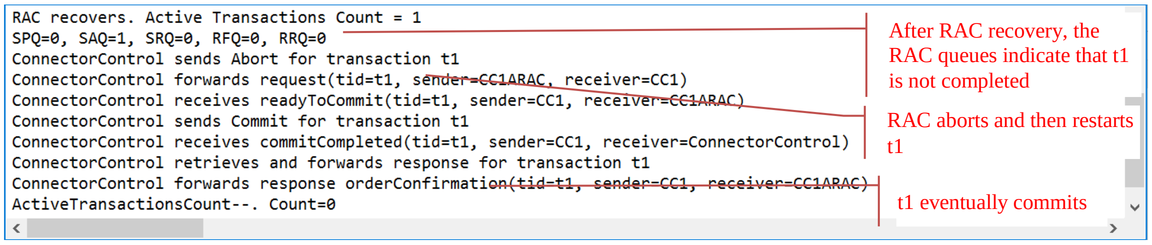

```
RAC recovers. Active Transactions Count = 1
SPQ=0, SAQ=1, SRQ=0, RFQ=0, RRQ=0
ConnectorControl sends Abort for transaction t1
ConnectorControl forwards request(tid=t1, sender=CC1ARAC, receiver=CC1)
ConnectorControl receives readyToCommit(tid=t1, sender=CC1, receiver=CC1ARAC)
ConnectorControl sends Commit for transaction t1
ConnectorControl receives commitCompleted(tid=t1, sender=CC1, receiver=ConnectorControl)
ConnectorControl retrieves and forwards response for transaction t1
ConnectorControl forwards response orderConfirmation(tid=t1, sender=CC1, receiver=CC1ARAC)
ActiveTransactionsCount--. Count=0
```
After RAC recovery, the RAC queues indicate that t1 is not completed
RAC aborts and then restarts t1
t1 eventually commits

Figure 10.20 Execution trace during recovery of RAC after a run-time failure

10.7 **Experimentation with the Reusable RAC**

This section describes the validation of the reusable RAC described in chapter 8. In this experiment, the Delivery Order Service (DOS) may receive both distributed and non-distributed transactions. Thus, these experiments are example of concurrent patterns integration of the general SOA pattern (see chapter 3 section 3.3.4) and the distributed transaction pattern (see chapter 3 section 3.3.5). As a result, the reusable RAC is needed to manage these different patterns as explained in chapter 8.

10.7.1 Self-Healing Scenario in the Online Shopping System

The failure scenario demonstrates the ability of the reusable RAC to recover failed transactions. In this scenario, the Delivery Order Service (DOS) is concurrently processing four transactions, which are in different states at the time of failure, as described below. Furthermore, two of these transactions are distributed while two of these transactions are non-distributed as follows:

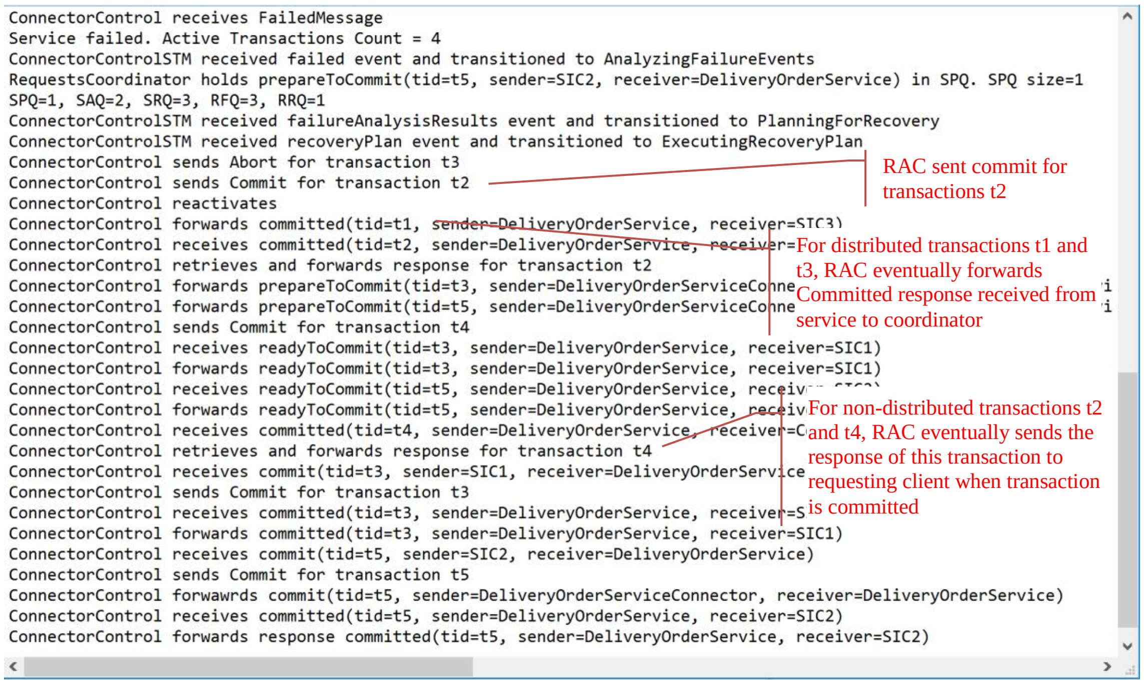
- Transaction t4: this transaction is non-distributed initiated by a Customer Coordinator (CC) that failed after the reusable RAC has received the Ready To Commit message from the DOS but before the RAC forwarded Commit to the DOS.
- Transaction t3: this transaction is distributed initiated by a Supplier Interaction Coordinator (SIC) that failed after the reusable RAC has sent the Prepare to Commit message to the service but before this RAC has received the Ready To Commit message from the service.
- Transactions t2: this transaction is non-distributed that failed after the reusable RAC has forwarded the Commit message to the service but before the reusable RAC received the Committed response.
- Transaction t1: this transaction is distributed that failed after the reusable RAC has received the Committed message from the service but before the RAC forwarded this response to the Supplier Interaction Coordinator (SIC) that initiated this distributed transaction.

As a result, the expected behavior in this scenario includes:

1. The reusable RAC receives a failure notification from the Recovery and Adaptation Manager.
2. The reusable RAC determines the appropriate recovery actions to recover the failed transactions t1-t4 as follows: (1) send Abort for transaction t3 and then

restart this transaction with the service when it is recovered, (2) send Commit for transaction t4, (3) resend Commit for transaction t2, and (4) send Committed for transaction t1 to the coordinator of this distributed transaction.

3. The reusable RAC receives the Reactivate command from the Recovery and Adaptation Manager to resume sending messages to the recovered DOS. As a result, DOS RAC sends the appropriate recovery actions to the recovered DOS as shown in (2).



```
ConnectorControl receives FailedMessage
Service failed. Active Transactions Count = 4
ConnectorControlSTM received failed event and transitioned to AnalyzingFailureEvents
RequestsCoordinator holds prepareToCommit(tid=t5, sender=SIC2, receiver=DeliveryOrderService) in SPQ. SPQ size=1
SPQ=1, SAQ=2, SRQ=3, RFQ=3, RRQ=1
ConnectorControlSTM received failureAnalysisResults event and transitioned to PlanningForRecovery
ConnectorControlSTM received recoveryPlan event and transitioned to ExecutingRecoveryPlan
ConnectorControl sends Abort for transaction t3
ConnectorControl sends Commit for transaction t2
ConnectorControl reactivates
ConnectorControl forwards committed(tid=t1, sender=DeliveryOrderService, receiver=SIC3)
ConnectorControl receives committed(tid=t2, sender=DeliveryOrderService, receiver=
ConnectorControl retrieves and forwards response for transaction t2
ConnectorControl forwards prepareToCommit(tid=t3, sender=DeliveryOrderServiceConne
ConnectorControl forwards prepareToCommit(tid=t5, sender=DeliveryOrderServiceConne
ConnectorControl sends Commit for transaction t4
ConnectorControl receives readyToCommit(tid=t3, sender=DeliveryOrderService, receiver=SIC1)
ConnectorControl forwards readyToCommit(tid=t3, sender=DeliveryOrderService, receiver=SIC1)
ConnectorControl receives readyToCommit(tid=t5, sender=DeliveryOrderService, receiv
ConnectorControl forwards readyToCommit(tid=t5, sender=DeliveryOrderService, receiv
ConnectorControl receives committed(tid=t4, sender=DeliveryOrderService, receiver=C
ConnectorControl retrieves and forwards response for transaction t4
ConnectorControl receives commit(tid=t3, sender=SIC1, receiver=DeliveryOrderService
ConnectorControl sends Commit for transaction t3
ConnectorControl receives committed(tid=t3, sender=DeliveryOrderService, receiver=S
ConnectorControl forwards committed(tid=t3, sender=DeliveryOrderService, receiver=SIC1)
ConnectorControl receives commit(tid=t5, sender=SIC2, receiver=DeliveryOrderService)
ConnectorControl sends Commit for transaction t5
ConnectorControl forwawrds commit(tid=t5, sender=DeliveryOrderServiceConnector, receiver=DeliveryOrderService)
ConnectorControl receives committed(tid=t5, sender=DeliveryOrderService, receiver=SIC2)
ConnectorControl forwards response committed(tid=t5, sender=DeliveryOrderService, receiver=SIC2)
```

RAC sent commit for transactions t2
For distributed transactions t1 and t3, RAC eventually forwards Committed response received from service to coordinator
For non-distributed transactions t2 and t4, RAC eventually sends the response of this transaction to requesting client when transaction is committed

Figure 10.21 Fragment of the execution trace of reusable RAC during planned failure scenario

When the reusable RAC received a failure notification indicating service failure, the contents of the various queues maintained by the RAC are as follows:

- The SPQ does not contain any request.
- The SAQ queues two Prepare To Commit requests that have been forwarded to the service:
    - PrepareToCommit(t3, SIC1, DOS, distributed)
    - PrepareToCommit(t4, CC3, DOS, non-distributed)
- The SRQ queues 3 messages of active transactions for which a response has been received by the RAC:
    - Commit(t2, DOSConnector, DOS, non-distributed)
    - PrepareToCommit(t1, SIC3, DOS, distributed)
    - Commit(t1, SIC3, DOS, distributed)
- The RFQ contains the following responses:
    - ReadyToCommit(t4, DOS, DOSConnector, non-distributed)
    - ReadyToCommit(t2, DOS, DOSConnector, non-distributed)
    - Committed(t1, DOS, SIC3, distributed)
- The RRQ queues one response as follows:
    - ReadyToCommit(t1, DOS, SIC3, distributed)

The trace also indicates that while the DOS is in the failed state, the DOS RAC received and held a new transaction t5 in the SPQ. During the analysis activity which is handled in the Analyzing Failure Events state (see Figure 3.6), the execution trace indicates that the RAC determined a status for each of these active transactions as follows:

- For transaction t3, the RAC determined the status of this transaction as *Preparing*, since the SAQ contains the Prepare To Commit request to the service but neither the RFQ nor the RRQ contain the Ready To Commit or Refuse To Commit response for this transaction.
- For transaction t4, the RAC determined the status of this transaction as *Prepared*, since the SAQ contains the Prepare To Commit request to the service and the RFQ contains the Ready To Commit response for this transaction.
- For transactions t2, the RAC determined the status of this transaction as *Committing*, since the SRQ contains the Commit request for this transaction but neither the RFQ nor the RRQ queues contain the Committed response for this transaction.
- For transaction t1, the RAC determined the status of this transaction as *Committed*, since the SRQ contains the Commit request to the service for this transaction and the RFQ contains a Committed response for this transaction.

During the planning activity which is handled in the Planning for Recovery state (Figure 3.6), the RAC determined recovery actions for each active transaction as follows:

- For transactions t3, since this transaction failed while being preparing to commit by the service during the first phase of 2PC, the recovery actions determined by the RAC for this transaction after service recovery were (1) to abort this transaction with the service and then (2) to restart this transaction with the recovered service.

- For transactions t4, since the service failed after preparing to commit this transaction, the recovery action determined by the RAC for this transaction was to send the Commit message to the recovered service.
- For transactions t2, since the service failed while committing this transaction, the recovery action determined by the RAC for this transaction was to resend the Commit message to the recovered service.
- For transaction t1, the action is to forward the Committed response queued in the RFQ of this transaction to the coordinator of this distributed transaction.

During the execution phase which is handled in the Executing Recovery Plan state, the reusable RAC recovered the requests of the transactions that must be restarted with the recovered service by moving these requests from the SAQ to the SPQ. In this scenario, only transaction t3 need to be restarted with the recovered service, since these transactions failed during phase 1 of the 2PC protocol. Therefore, the execution trace indicates that this message is moved from the SAQ to the SPQ. The content of SPQ after recovery of this transaction is:

- PrepareToCommit(t3, SIC1, DOS, distributed)
- PrepareToCommit(t5, SIC2, DOS, distributed)

When the RAC is reactivated after the service has recovered, the execution trace indicates that the RAC aborted and then restarted transaction t3 with the recovered service, (2) requested the recovered service to commit transactions t2 and t4, and (3) forwarded the Committed response of transaction t1 to the coordinator of the distributed transaction. The execution trace shows that transactions t1-t4 as well as held transaction t5 terminated normally.

10.7.2 Self-Configuration Scenario in the Online Shopping System

To illustrate the behavior of the reusable RAC during adaptation, I use an adaptation scenario that involves adapting the DOS service while it is concurrently engaged in transactions of different types. In this scenario, the DOS service is concurrently processing 4 transactions, two of which are distributed while the other two are non-distributed. As a result, the expected behavior in this scenario includes:

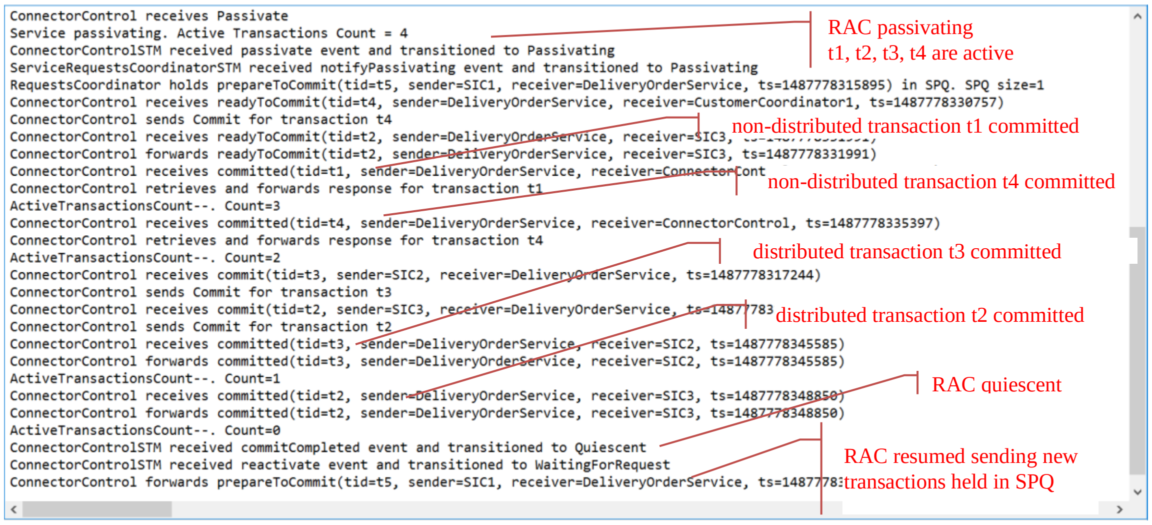
1. When the reusable RAC receives the Passivate command, the RAC transitions to the Passivating state in order to allow transactions t1-t4 to complete. Furthermore, the reusable RAC holds any new transactions in the SPQ until dynamic adaptation is completed.
2. When transactions t1-t4 complete, the reusable RAC transitions to the Quiescent state.
3. When dynamic service adaptation is completed, the reusable RAC receives the Reactivate command from the Recovery and Adaptation Manager. As a result, the RAC resumes sending messages to the adapted DOS including the transactions held in the SPQ.

In this scenario, the execution trace (Figure 10.22) indicates that when reusable RAC received the Passivate command, the content of the RAC's queues is as follows:

- The SPQ contains one transaction that was held due to dynamic service adaptation:
    - PrepareToCommit(t5, SIC1, DOS, distributed)

- The SAQ contains two prepare to commit requests sent to the service:
    - PrepareToCommit(t4, CC1, DOS, non-distributed)
    - PrepareToCommit(t2, SIC3, DOS, distributed)
- The SRQ contains two messages:
    - Commit(t1, DOSConnector, DOS, non-distributed)
    - PrepareToCommit(t3, SIC2, DOS, distributed)
- The RFQ contains one response that is not forwarded:
    - ReadyToCommit(t1, DOS, CC1, non-distributed)
- The RRQ contains one forwarded response:
    - ReadyToCommit(t3, DOS, SIC2, distributed)

As a result of passivation, the execution trace indicates that the reusable RAC transitioned into the Passivating state (Figure 10.22) where it permitted the four active transactions (t1-t4) to gradually terminate. After the DOS completed all active transactions, the execution trace indicates that the reusable RAC transitioned to the Quiescent state at which time the DOS was dynamically replaced by an updated version of the service. After adaptation is completed, the service RAC received the reactivate command. As a result, the RAC transitioned to the Active state and forwarded the held transaction t5 queued in its SPQ to the DOS. At this point, normal execution is resumed between RAC and the DOS, which corresponds to the expected outcome of this scenario.



```
ConnectorControl receives Passivate
Service passivating. Active Transactions Count = 4
ConnectorControlSTM received passivate event and transitioned to Passivating
ServiceRequestsCoordinatorSTM received notifyPassivating event and transitioned to Passivating
RequestsCoordinator holds prepareToCommit(tid=t5, sender=SIC1, receiver=DeliveryOrderService, ts=1487778315895) in SPQ. SPQ size=1
ConnectorControl receives readyToCommit(tid=t4, sender=DeliveryOrderService, receiver=CustomerCoordinator1, ts=1487778330757)
ConnectorControl sends Commit for transaction t4
ConnectorControl receives readyToCommit(tid=t2, sender=DeliveryOrderService, receiver=SIC3,
ConnectorControl forwards readyToCommit(tid=t2, sender=DeliveryOrderService, receiver=SIC3, ts=1487778331991)
ConnectorControl receives committed(tid=t1, sender=DeliveryOrderService, receiver=ConnectorCont
ConnectorControl retrieves and forwards response for transaction t1
ActiveTransactionsCount--. Count=3
ConnectorControl receives committed(tid=t4, sender=DeliveryOrderService, receiver=ConnectorControl, ts=1487778335397)
ConnectorControl retrieves and forwards response for transaction t4
ActiveTransactionsCount--. Count=2
ConnectorControl receives commit(tid=t3, sender=SIC2, receiver=DeliveryOrderService, ts=1487778317244)
ConnectorControl sends Commit for transaction t3
ConnectorControl receives commit(tid=t2, sender=SIC3, receiver=DeliveryOrderService, ts=14877783
ConnectorControl sends Commit for transaction t2
ConnectorControl receives committed(tid=t3, sender=DeliveryOrderService, receiver=SIC2, ts=1487778345585)
ConnectorControl forwards committed(tid=t3, sender=DeliveryOrderService, receiver=SIC2, ts=1487778345585)
ActiveTransactionsCount--. Count=1
ConnectorControl receives committed(tid=t2, sender=DeliveryOrderService, receiver=SIC3, ts=1487778348850)
ConnectorControl forwards committed(tid=t2, sender=DeliveryOrderService, receiver=SIC3, ts=1487778348850)
ActiveTransactionsCount--. Count=0
ConnectorControlSTM received commitCompleted event and transitioned to Quiescent
ConnectorControlSTM received reactivate event and transitioned to WaitingForRequest
ConnectorControl forwards prepareToCommit(tid=t5, sender=SIC1, receiver=DeliveryOrderService, ts=14877783
```

RAC passivating
t1, t2, t3, t4 are active
non-distributed transaction t1 committed
non-distributed transaction t4 committed
distributed transaction t3 committed
distributed transaction t2 committed
RAC quiescent
RAC resumed sending new transactions held in SPQ

Figure 10.22 Fragment of the execution trace of reusable RAC during planned adaptation scenario

10.7.3 Self-Healing Scenario in the Emergency Monitoring System

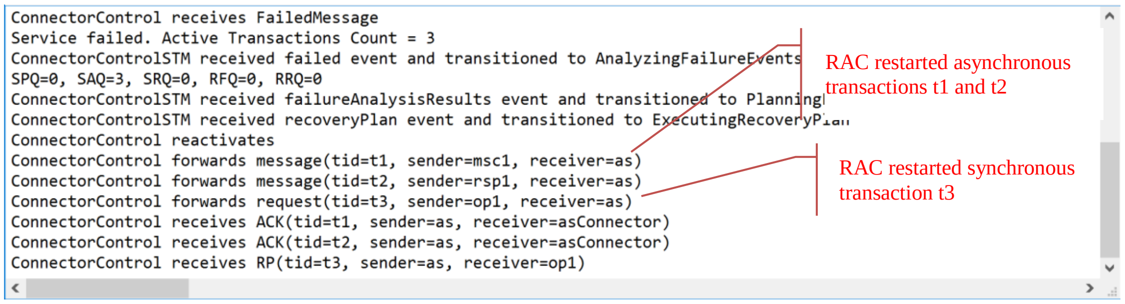
This failure scenario demonstrates the ability of the reusable RAC to recover failed transactions of different patterns in the EMS. In this scenario, the Alarm Service (AS) is concurrently processing three transactions. Two of these transactions (t1 and t2) are asynchronous while one transaction (t3) is synchronous. The expected behavior in this scenario includes:

1. The AS RAC receives a failure notification from the Recovery and Adaptation Manager (RAM).
2. The AS RAC determines a recovery action to recover the failed transactions t1-t3.
3. The AS RAC receives the Reactivate command from the Recovery and Adaptation Manager to resume sending messages to the recovered AS. As a result, AS RAC resends transactions t1-t3 to the recovered AS.

In this scenario, the execution trace (Figure 10.23) indicates that the SAQ contains three messages that have been forwarded to the AS:

- Message(t1, MSC1, AS)
- Message(t2, RSP1, AS)
- Request(t3, OP1, AS)

During the analysis activity which is handled in the Analyzing Failure Events state, the execution trace indicates that the AS RAC determined that transactions *t1-t3* as failed since AS failure occurred before the AS RAC received the corresponding ACK messages for transactions t1-t2 and the response for transaction t3. The execution trace indicates that the AS RAC moved these transactions from the SAQ to the SPQ. When the RAC is reactivated after the AS has recovered, the execution trace indicates that the RAC sent lost transactions t1-t3 to the recovered AS. Furthermore, the execution trace indicates that the RAC eventually received ACK messages for transactions t1-t2 and a response for transaction t3 from the recovered AS, which corresponds to what is expected in this experiment.



```
ConnectorControl receives FailedMessage
Service failed. Active Transactions Count = 3
ConnectorControlSTM received failed event and transitioned to AnalyzingFailureEvents
SPQ=0, SAQ=3, SRQ=0, RFQ=0, RRQ=0
ConnectorControlSTM received failureAnalysisResults event and transitioned to PlanningR
ConnectorControlSTM received recoveryPlan event and transitioned to ExecutingRecoveryPlan
ConnectorControl reactivates
ConnectorControl forwards message(tid=t1, sender=msc1, receiver=as)
ConnectorControl forwards message(tid=t2, sender=rsp1, receiver=as)
ConnectorControl forwards request(tid=t3, sender=op1, receiver=as)
ConnectorControl receives ACK(tid=t1, sender=as, receiver=asConnector)
ConnectorControl receives ACK(tid=t2, sender=as, receiver=asConnector)
ConnectorControl receives RP(tid=t3, sender=as, receiver=op1)
```

RAC restarted asynchronous transactions t1 and t2
RAC restarted synchronous transaction t3

Figure 10.23 Fragment of the execution trace of reusable RAC during planned failure scenario in the EMS

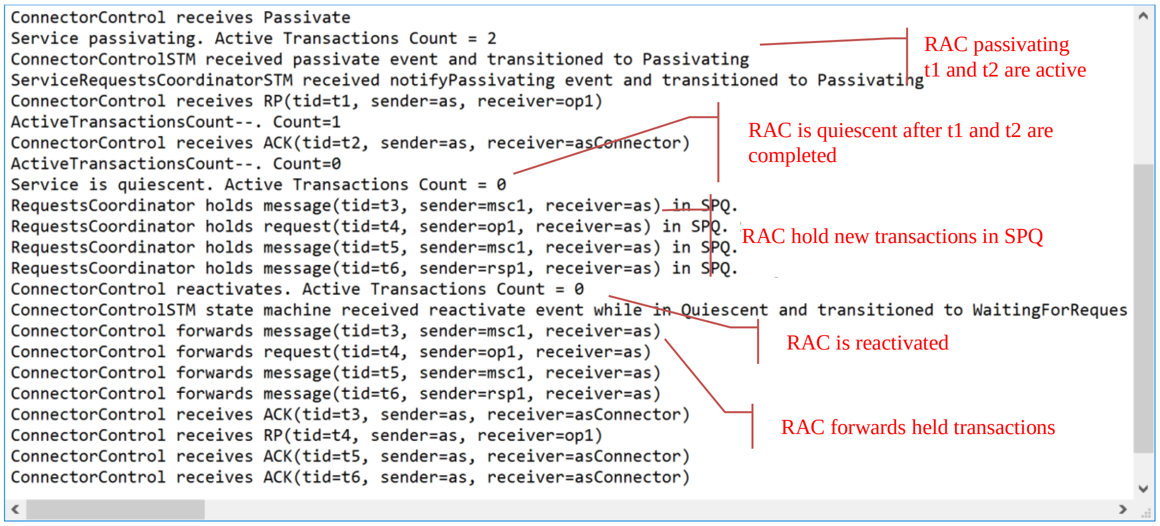
10.7.4 Self-Configuration Scenario in the Emergency Monitoring System

To illustrate the behavior of the reusable RAC during adaptation of a component that integrates multiple patterns, I use an adaptation scenario that involves adapting the Alarm Service (AS) component. In this scenario, the AS is concurrently processing two transactions. Transaction t1 is synchronous while transaction t2 is asynchronous. The expected behavior in this scenario includes:

1. When the AS RAC receives the Passivate command, the RAC transitions to the Passivating state until it receives a response from AS for transaction t1 and an ACK for transaction t2. Furthermore, the AS RAC holds any new transactions to the AS in the SPQ until dynamic adaptation is completed.
2. The AS RAC transitions to the Quiescent state in which the AS can be safely adapted.
3. The AS RAC receives a Reactivate command from the Recovery and Adaptation Manager so that the RAC resumes sending messages to the adapted AS. As a result, the AS RAC sends any held transactions in the SPQ to the adapted AS.

In this scenario, the execution trace indicates that when the AS RAC received the Passivate command, the RAC transitioned into the Passivating state (Figure 10.24) until the AS sends a response for transaction t1 and an ACK message for transaction t2. After the AS has completed all transactions that it is currently engaged in, the execution trace indicates that the AS RAC transitioned to the Quiescent state during which the AS was dynamically replaced. While in the Quiescent state, the AS RAC received three new asynchronous transactions and one synchronous transaction. The execution trace indicates that the RAC queued these new transactions into the Service Pending Queue.

After adaptation is completed, the AS RAC received the reactivate command. At this point, normal execution is resumed between RAC and the adapted AS. The execution trace indicates that the AS RAC forwarded all transactions held previously in the pending queue due to dynamic adaptation to the adapted AS. Furthermore, the execution trace indicates that the RAC eventually received from the adapted AS an ACK message for each forwarded asynchronous transaction and a response for the forwarded synchronous transaction, which corresponds to what is expected in this experiment.



```
ConnectorControl receives Passivate
Service passivating. Active Transactions Count = 2
ConnectorControlSTM received passivate event and transitioned to Passivating
ServiceRequestsCoordinatorSTM received notifyPassivating event and transitioned to Passivating
ConnectorControl receives RP(tid=t1, sender=as, receiver=op1)
ActiveTransactionsCount--. Count=1
ConnectorControl receives ACK(tid=t2, sender=as, receiver=asConnector)
ActiveTransactionsCount--. Count=0
Service is quiescent. Active Transactions Count = 0
RequestsCoordinator holds message(tid=t3, sender=msc1, receiver=as) in SPQ.
RequestsCoordinator holds request(tid=t4, sender=op1, receiver=as) in SPQ.
RequestsCoordinator holds message(tid=t5, sender=msc1, receiver=as) in SPQ.
RequestsCoordinator holds message(tid=t6, sender=rsp1, receiver=as) in SPQ.
ConnectorControl reactivates. Active Transactions Count = 0
ConnectorControlSTM state machine received reactivate event while in Quiescent and transitioned to WaitingForReques
ConnectorControl forwards message(tid=t3, sender=msc1, receiver=as)
ConnectorControl forwards request(tid=t4, sender=op1, receiver=as)
ConnectorControl forwards message(tid=t5, sender=msc1, receiver=as)
ConnectorControl forwards message(tid=t6, sender=rsp1, receiver=as)
ConnectorControl receives ACK(tid=t3, sender=as, receiver=asConnector)
ConnectorControl receives RP(tid=t4, sender=as, receiver=op1)
ConnectorControl receives ACK(tid=t5, sender=as, receiver=asConnector)
ConnectorControl receives ACK(tid=t6, sender=as, receiver=asConnector)
```

RAC passivating
t1 and t2 are active
RAC is quiescent after t1 and t2 are completed
RAC hold new transactions in SPQ
RAC is reactivated
RAC forwards held transactions

Figure 10.24 Fragment of the execution trace of reusable RAC during planned adaptation scenario in the EMS

11 CONCLUSION AND FUTURE WORK

This dissertation has investigated a reuse, model-based approach for self-healing and self-configuration in component-based software architectures. This dissertation described how connectors in component-based software architectures (CBSAs) can be extended with recovery and adaptation capabilities to assist in self-healing and self-configuration. Furthermore, this dissertation described the design of DARE, an architecture-based, decentralized framework that provides both self-healing and self-configuration properties to CBSAs. The design of DARE is based on a decentralized MAPE-K loop model in which DARE carries out recovery and dynamic adaptation when only partial knowledge of the software system is known to each node.

The contributions of this dissertation are as follows:

1. *Design of recovery and adaptation connectors for various SOA patterns*. This dissertation described the design of recovery and adaptation connectors that handle recovery and adaptation concerns of clients, coordinators and services in SOA patterns.
2. *Design of recovery and adaptation connectors for various asynchronous patterns*. This dissertation described the design of recovery and adaptation connectors that handle recovery and adaptation concerns of producers and consumers in asynchronous patterns.

3. *Design of a decentralized, self-healing, and self-configuration framework*. This dissertation described the design of DARE, an architecture-based and decentralized framework for providing large and highly dynamic CBSAs with self-healing and self-configuration properties.
4. *Design of a reusable recovery and adaptation connector that supports integration of patterns*. This dissertation described how variability in recovery and adaptation connectors can be managed using the software product line technology in which a reusable connector can handle integration of multiple architectural patterns.
5. *Formal properties of the approach.* This dissertation defined several properties that are ensured by the DARE framework and RACs during normal execution, failure recovery, and dynamic adaptation.
6. *Experimental Validation of the approach*. This dissertation contains experimental validation and results of both the DARE framework and RACs which show their capabilities during failure recovery and dynamic adaptation.

There are several directions for future work. In this dissertation, DARE integrates two of the MAPE-K self-* properties: self-healing and self-configuration properties. One open challenge is to consider how two other MAPE-K self-* properties, self-optimization and self-protection properties, can be integrated into DARE as well. For instance, the DARE framework currently selects any random healthy node to host recovered components and connectors. Another alternative is to incorporate self-optimization techniques so that DARE can select the most optimal node for hosting these components and connectors. With respect to self-protection, DARE currently assumes that failures are not caused by malicious attacks. However, a self-protection approach can be considered to relax this assumption.

This research has considered nodes that fail according to the fail-stop assumption in which components do not behave erroneously but simply cease functioning when they fail. Future work includes expanding the approach to handle network link failures that may cause the network to partition into several disjoint networks. Other future works include: investigating failure recovery of the DARE framework during dynamic adaptation or recovery, investigating recovery and adaptation patterns of more architectural patterns, analyzing the performance of DARE, considering real-time issues, and scaling up to large distributed software systems.

REFERENCES

Alvisi, L., Marzullo, K., 1998. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. IEEE Trans Softw Eng 24, 149–159. doi:10.1109/32.666828

Angarita, R., Rukoz, M., Cardinale, Y., 2016. Modeling dynamic recovery strategy for composite web services execution. World Wide Web 19, 89–109. doi:10.1007/s11280-015-0329-1

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1, 11–33. doi:10.1109/TDSC.2004.2

Baresi, L., Guinea, S., Pasquale, L., 2012. Service-Oriented Dynamic Software Product Lines. Computer 45, 42–48. doi:10.1109/MC.2012.289

Bencomo, N., Hallsteinsen, S., Santana de Almeida, E., 2012. A View of the Dynamic Software Product Line Landscape. Computer 45, 36–41. doi:10.1109/MC.2012.292

Bernstein, P.A., Hsu, M., Mann, B., 1990. Implementing Recoverable Requests Using Queues, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, SIGMOD '90. ACM, New York, NY, USA, pp. 112–122. doi:10.1145/93597.98721

Bernstein, P.A., Newcomer, E., 2009. Principles of Transaction Processing, Second Edition, 2 edition. ed. Morgan Kaufmann, Burlington, MA.

Bisadi, M., Sharifi, M., 2009. Component-Based Self-Healing via Cellular Adaptation, in: Autonomic and Autonomous Systems, International Conference on. IEEE Computer Society, Los Alamitos, CA, USA, pp. 75–81. doi:10.1109/ICAS.2009.61

Bosch, J., Capilla, R., 2012. Dynamic Variability in Software-Intensive Embedded System Families. Computer 45, 28–35. doi:10.1109/MC.2012.287

Bruning, S., Weissleder, S., Malek, M., 2007. A Fault Taxonomy for Service-Oriented Architecture, in: 10th IEEE High Assurance Systems Engineering Symposium,

2007. HASE ’07. Presented at the 10th IEEE High Assurance Systems Engineering Symposium, 2007. HASE ’07, pp. 367–368. doi:10.1109/HASE.2007.46

Candea, G., C, G., Kiciman, E., Zhang, S., Fox, A., Keyani, P., Fox, O., 2003. JAGR: An Autonomous Self-Recovering Application Server.

Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., Snir, M., 2014. Toward Exascale Resilience: 2014 Update. Supercomput. Front. Innov. 1, 1–28.

Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Presti, F.L., Mirandola, R., 2012. MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems. IEEE Trans. Softw. Eng. 38, 1138–1159. doi:10.1109/TSE.2011.68

Cheng, S.-W., Garlan, D., 2007. Handling Uncertainty in Autonomic Systems.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns, 3rd edition. ed. Addison-Wesley Professional, Boston.

Danilecki, A., Hołenko, M., Kobusińska, A., Szychowiak, M., Zierhoffer, P., 2013. Applying Message Logging to Support Fault-Tolerance of SOA Systems. Found. Comput. Decis. Sci. 38, 145–158. doi:10.2478/fcds-2013-0006

Danilecki, A., Hołenko, M., Kobusińska, A., Szychowiak, M., Zierhoffer, P., 2011. ReServE Service: An Approach to Increase Reliability in Service Oriented Systems, in: Malyshkin, V. (Ed.), Parallel Computing Technologies, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 244–256. doi:10.1007/978-3-642-23178-0_22

Dashofy, E.M., van der Hoek, A., Taylor, R.N., 2002. Towards Architecture-based Self-healing Systems, in: Proceedings of the First Workshop on Self-Healing Systems, WOSS ’02. ACM, New York, NY, USA, pp. 21–26. doi:10.1145/582128.582133

David M Holman, D.S.L., 2008. A Survey of Routing Techniques in Store-and-Forward and Wormhole Interconnects.

Elnozahy, E.N. (Mootaz), Alvisi, L., Wang, Y.-M., Johnson, D.B., 2002. A Survey of Rollback-recovery Protocols in Message-passing Systems. ACM Comput Surv 34, 375–408. doi:10.1145/568522.568525

Ericson, C., 1999. Fault Tree Analysis - A History, in: Proceedings of the 17th International Systems Safety Conference.

Freiling, F.C., Guerraoui, R., Kuznetsov, P., 2011. The Failure Detector Abstraction. ACM Comput Surv 43, 9:1–9:40. doi:10.1145/1883612.1883616

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software, 1 edition. ed. Addison-Wesley Professional, Reading, Mass.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P., 2004. Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer 37, 46–54. doi:10.1109/MC.2004.175

Garlan, D., Schmerl, B., 2002. Model-based Adaptation for Self-healing Systems, in: Proceedings of the First Workshop on Self-Healing Systems, WOSS '02. ACM, New York, NY, USA, pp. 27–32. doi:10.1145/582128.582134

Gomaa, H., 2011. Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures, 1 edition. ed. Cambridge University Press, Cambridge ; New York.

Gomaa, H., 2004. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional, Boston.

Gomaa, H., Hashimoto, K., 2012. Dynamic Self-adaptation for Distributed Service-oriented Transactions, in: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '12. IEEE Press, Piscataway, NJ, USA, pp. 11–20.

Gomaa, H., Hashimoto, K., 2011. Dynamic Software Adaptation for Service-oriented Product Lines, in: Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11. ACM, New York, NY, USA, p. 35:1–35:8. doi:10.1145/2019136.2019176

Gomaa, H., Hashimoto, K., Kim, M., Malek, S., Menascé, D.A., 2010. Software Adaptation Patterns for Service-oriented Architectures, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10. ACM, New York, NY, USA, pp. 462–469. doi:10.1145/1774088.1774185

Gomaa, H., Hussein, M., 2004. Software reconfiguration patterns for dynamic evolution of software architectures, in: Fourth Working IEEE/IFIP Conference on Software Architecture, 2004. WICSA 2004. Proceedings. Presented at the Fourth Working IEEE/IFIP Conference on Software Architecture, 2004. WICSA 2004. Proceedings, pp. 79–88. doi:10.1109/WICSA.2004.1310692

Guerraoui, R., Schiper, A., 1997. Software-Based Replication for Fault Tolerance. Computer 30, 68–74. doi:10.1109/2.585156

Hallsteinsen, S., Geihs, K., Paspallis, N., Eliassen, F., Horn, G., Lorenzo, J., Mamelli, A., Papadopoulos, G.A., 2012. A development framework and methodology for self-adapting applications in ubiquitous computing environments. J. Syst. Softw., Self-Adaptive Systems 85, 2840–2859. doi:10.1016/j.jss.2012.07.052

Hinchey, M., Park, S., Schmid, K., 2012. Building Dynamic Software Product Lines. Computer 45, 22–26. doi:10.1109/MC.2012.332

Huebscher, M.C., McCann, J.A., 2008. A Survey of Autonomic Computing-Degrees, Models, and Applications. ACM Comput Surv 40, 7:1–7:28. doi:10.1145/1380584.1380585

Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36, 41–50. doi:10.1109/MC.2003.1160055

Kramer, J., Magee, J., 2007. Self-Managed Systems: an Architectural Challenge, in: Future of Software Engineering, 2007. FOSE '07. Presented at the Future of Software Engineering, 2007. FOSE '07, pp. 259–268. doi:10.1109/FOSE.2007.19

Kramer, J., Magee, J., 1990. The evolving philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. 16, 1293–1306. doi:10.1109/32.60317

Krupitzer, C., Roth, F.M., VanSyckel, S., Schiele, G., Becker, C., 2015. A Survey on Engineering Approaches for Self-adaptive Systems. Pervasive Mob Comput 17, 184–206. doi:10.1016/j.pmcj.2014.09.009

Lamport, L., 1978. Time, Clocks, and the Ordering of Events in a Distributed System. Commun ACM 21, 558–565. doi:10.1145/359545.359563

Lee, B., Park, T., Yeom, H.Y., Cho, Y., 1998. An efficient algorithm for causal message logging, in: Seventeenth IEEE Symposium on Reliable Distributed Systems, 1998. Proceedings. Presented at the Seventeenth IEEE Symposium on Reliable Distributed Systems, 1998. Proceedings, pp. 19–25. doi:10.1109/RELDIS.1998.740470

Lemos, R. de, Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Göschka, K.M., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezzè, M., Prehofer, C., Schäfer, W., Schlichting, R., Smith, D.B., Sousa, J.P., Tahvildari, L., Wong, K., Wuttke, J., 2013. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap, in: Lemos, R. de, Giese, H., Müller, H.A., Shaw, M. (Eds.), Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–32.

Leveson, N.G., Harvey, P.R., 1983. Analyzing Software Safety. IEEE Trans. Softw. Eng. SE-9, 569–579. doi:10.1109/TSE.1983.235116

Li, G., Han, Y., Zhao, Z., Wang, J., Wagner, R.M., 2006. Facilitating Dynamic Service Compositions by Adaptable Service Connectors: Int. J. Web Serv. Res. 3, 68–84. doi:10.4018/jwsr.2006010104

Lomet, D., Tuttle, M., 2003. A Theory of Redo Recovery, in: Proceedings of the 2003 ACM SIGMOD Conference on Management of Data. pp. 397–406.

Magalhães, J.P., Silva, L.M., 2015. SHÕWA: A Self-Healing Framework for Web-Based Applications. ACM Trans Auton Adapt Syst 10, 4:1–4:28. doi:10.1145/2700325

Menasce, D., Gomaa, H., Malek, S., Sousa, J.P., 2011. SASSY: A Framework for Self-Architecting Service-Oriented Systems. IEEE Softw. 28, 78–85. doi:10.1109/MS.2011.22

Menasce, D.A., Sousa, J.P., Malek, S., Gomaa, H., 2010. Qos Architectural Patterns for Self-architecting Software Systems, in: Proceedings of the 7th International Conference on Autonomic Computing, ICAC ’10. ACM, New York, NY, USA, pp. 195–204. doi:10.1145/1809049.1809084

Modafferi, S., Conforti, E., 2006. Methods for Enabling Recovery Actions in Ws-BPEL, in: Proceedings of the 2006 Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I, ODBASE’06/OTM’06. Springer-Verlag, Berlin, Heidelberg, pp. 219–236. doi:10.1007/11914853_14

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P., 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. ACM Trans Database Syst 17, 94–162. doi:10.1145/128765.128770

Neti, S., Muller, H.A., 2007. Quality Criteria and an Analysis Framework for Self-Healing Systems, in: International Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2007. ICSE Workshops SEAMS ’07. Presented at the International Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2007. ICSE Workshops SEAMS ’07, pp. 6–6. doi:10.1109/SEAMS.2007.15

Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L., 1999. An Architecture-Based Approach to Self-Adaptive Software. IEEE Intell. Syst. 14, 54–62. doi:10.1109/5254.769885

Porter, J., Menasce, D., Gomaa, H., 2016. DeSARM: A Decentralized Software Architecture Discovery Mechanism for Distributed Systems. Presented at the 11th International Workshop on Models@run.time (MODELS 2016), Saint-Malo, France.

Psaier, H., Dustdar, S., 2010. A survey on self-healing systems: approaches and systems. Computing 91, 43–73. doi:10.1007/s00607-010-0107-y

Ramirez, A.J., Cheng, B.H.C., 2010. Design Patterns for Developing Dynamically Adaptive Systems, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS ’10. ACM, New York, NY, USA, pp. 49–58. doi:10.1145/1808984.1808990

Rauzy, A., 1993. New algorithms for fault trees analysis. Reliab. Eng. Syst. Saf. 40, 203–211. doi:10.1016/0951-8320(93)90060-C

Raynal, M., 1992. About Logical Clocks for Distributed Systems. SIGOPS Oper Syst Rev 26, 41–48. doi:10.1145/130704.130708

Salatge, N., Fabre, J.-C., 2007. Fault Tolerance Connectors for Unreliable Web Services, in: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN ’07. Presented at the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN ’07, pp. 51–60. doi:10.1109/DSN.2007.48

Salehie, M., Tahvildari, L., 2009. Self-adaptive Software: Landscape and Research Challenges. ACM Trans Auton Adapt Syst 4, 14:1–14:42. doi:10.1145/1516533.1516538

Sawyer, P., Mazo, R., Diaz, D., Salinesi, C., Hughes, D., 2012. Using Constraint Programming to Manage Configurations in Self-Adaptive Systems. Computer 45, 56–63. doi:10.1109/MC.2012.286

Schneider, C., Barker, A., Dobson, S., 2015. A survey of self-healing systems frameworks. Softw. Pract. Exp. 45, 1375–1398. doi:10.1002/spe.2250

Schroeder, B., Gibson, G.A., 2007. Understanding failures in petascale computers. J. Phys. Conf. Ser. 78, 012022. doi:10.1088/1742-6596/78/1/012022

Silva, L.M., Alonso, J., Torres, J., 2009. Using Virtualization to Improve Software Rejuvenation. IEEE Trans. Comput. 58, 1525–1538. doi:10.1109/TC.2009.119

Sommerville, I., 2010. Software Engineering, 9th ed. Addison-Wesley Publishing Company, USA.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H., 2003. Chord: a scalable peer-to-peer lookup protocol for Internet applications. IEEEACM Trans. Netw. 11, 17–32. doi:10.1109/TNET.2002.808407

Stojnic, N., Schuldt, H., 2012. OSIRIS-SR: A Safety Ring for self-healing distributed composite service execution, in: 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). Presented at the 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 21–26. doi:10.1109/SEAMS.2012.6224387

Strom, R., Yemini, S., 1985. Optimistic Recovery in Distributed Systems. ACM Trans Comput Syst 3, 204–226. doi:10.1145/3959.3962

Subramanian, S., Thiran, P., Narendra, N.C., Mostefaoui, G.K., Maamar, Z., 2008. On the Enhancement of BPEL Engines for Self-Healing Composite Web Services, in: International Symposium on Applications and the Internet, 2008. SAINT 2008. Presented at the International Symposium on Applications and the Internet, 2008. SAINT 2008, pp. 33–39. doi:10.1109/SAINT.2008.12

Tanenbaum, A.S., Steen, M.V., 2006. Distributed Systems: Principles and Paradigms, 2 edition. ed. Prentice Hall, Upper Saddle RIiver, NJ.

Taylor, R.N., Medvidovic, N., Dashofy, E.M., 2009. Software Architecture: Foundations, Theory, and Practice, 1 edition. ed. Wiley, Hoboken, NJ.

Vandewoude, Y., Ebraert, P., Berbers, Y., D’Hondt, T., 2007. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. IEEE Trans. Softw. Eng. 33, 856–868. doi:10.1109/TSE.2007.70733

Vossen, G., Weikum, G., 2001. Transactional Information Systems. Morgan Kaufmann.

Wang, R., Salzberg, B., Lomet, D., 2007. Log-based Recovery for Middleware Servers, in: Proceedings of the 2007 ACM SIGMOD International Conference on

Management of Data, SIGMOD ’07. ACM, New York, NY, USA, pp. 425–436. doi:10.1145/1247480.1247528

Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M., 2013. On Patterns for Decentralized Control in Self-Adaptive Systems, in: Lemos, R. de, Giese, H., Müller, H.A., Shaw, M. (Eds.), Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 76–107. doi:10.1007/978-3-642-35813-5_4

BIOGRAPHY

Emad Yousif Albassam received his Bachelor of Computer Science from King Abdulaziz University, Jeddah, Saudi Arabia in 2007. In 2012, he graduated from George Mason University, VA, USA with a Master of Science in Software Engineering. After that, he started his Doctoral degree in Information Technology at George Mason University. During his graduate studies, he published several research papers. Emad is also a faculty member in the Deanship of Information Technology at King Abdulazuz University, Jeddah, Saudi Arabia.