A Planning-Based Approach to Failure Recovery in Distributed Systems

by

Naveed Arshad

B.S., Ghulam Ishaq Khan Institute of Engineering Sciences

and Technology, Pakistan, 1999

M.S., University of Colorado at Boulder, USA, 2003

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2006

This thesis entitled:
A Planning-Based Approach to Failure Recovery in Distributed Systems
written by Naveed Arshad
has been approved for the Department of Computer Science

Professor Alexander L. Wolf

Professor Dennis M. Heimbigner

Date ____________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Arshad, Naveed (Ph.D., Computer Science)

A Planning-Based Approach to Failure Recovery in Distributed Systems

Thesis directed by Professor Alexander L. Wolf

Automated failure recovery in distributed systems poses a tough challenge because of myriad requirements and dependencies among its components. Moreover, failure scenarios are usually unpredictable so they cannot easily be foreseen. Therefore, it is not practical to enumerate all possible failure scenarios and a way to recover a distributed system for each of them. Due to this reason, present failure recovery techniques are highly manual and have considerable downtime associated with them. In this dissertation, we have developed a planning-based approach to automated failure recovery in distributed component-based systems. This approach automates failure recovery through continuous monitoring of the system. Therefore, an exact system state is always available with a failure monitor. When a failure is detected the monitor performs various checks to ensure that it is not a false positive or false negative. A dependency analyzer then checks effects of the failure on other parts of the system. After this an offline planning procedure is performed to take the system from a failed state to a working state. This planning is performed using an artificially intelligent (AI) planner. By using planning, this approach can be used to recover from a variety of failed states and reach any of several acceptable states: from minimal functionality to complete recovery. When a plan is calculated, it is executed onto the system to bring it back to a working state. We have evaluated this technique through various online and synthetic experiments performed on various distributed applications. Our results have shown that this is indeed an effective technique to automatically recover component-based distributed systems from a failure. Our results have also shown that this technique can also scale to large-scale distributed systems.

Dedication

All Praise and thanks be to Allah! We praise and thank Him, ask Him for His Help and Forgiveness, and we seek refuge in Allah from the evils of our souls and the mischiefs of our deeds. He whom Allah guides will not be misled, and he whom Allah misleads will never have a guide. I testify that there is no deity but Allah alone, without any partners, and that Muhammad is His slave servant and Messenger.

This dissertation is a humble effort to augment the body of knowledge in the field of computer science particularly software engineering. Like all the things in this world mistakes and flaws are possible in it. I wholeheartedly accept all the mistakes and flaws that I have made in this dissertation and attribute all the beneficial knowledge in it to Allah almighty. An average person like me could never complete a PhD dissertation unless helped by Allah. I would like to thank Allah almighty for helping me in every aspect of this dissertation.

Acknowledgements

I would like to thank my advisors Professor Alexander L. Wolf and Professor Dennis M. Heimbigner for all their support and help during all these five years I spent at the University of Colorado. Moreover, I would like to thank other members of the Software Engineering Research Group (SERL) for their continuous feedback on my research. I would also like to thank Professor Amer Diwan, Professor Ken Anderson and Professor Jim Martin for their help during my time here at University of Colorado.

My very special thanks to my parents, my wife, my children, my extended family and my friends for everything that they did for me during this time. Their encouragement, support and help was always there for me whenever I needed it.

Contents

Chapter

1 Introduction 1

1.1 Failure Recovery Techniques in Distributed Systems . . . 3

1.1.1 Redundancy-based Failure Recovery . . . 3

1.1.2 Failure Recovery Scripts . . . 4

1.2 Approach to Automating Failure Recovery . . . 5

1.3 Planning and Failure Recovery . . . 6

1.4 Contributions and their Evaluations . . . 7

1.5 Scope and Assumptions . . . 8

1.6 Organization of the Dissertation . . . 9

2 Foundational and Related Work 11

2.1 Foundational Work . . . 11

2.1.1 Dynamic Reconfiguration . . . 11

2.1.2 Planning . . . 21

2.1.3 Dependency Management . . . 31

2.1.4 Fault Tolerance . . . 36

2.2 Related Work . . . 39

2.2.1 Control Theory-Based Techniques . . . 39

2.2.2 Architecture-Based Recovery . . . 44

2.2.3 Recovery-Oreinted Computing 47
2.2.4 Miscellaneous Approaches to Failure Recovery 50

3 Motivating Example 53
3.1 Reasons of Failures 53
3.2 A Typical Internet-Service System 54
3.3 Distributed Systems Architecture 58

4 Modeling for Failure Recovery 61
4.1 Application Model 63
4.1.1 Application Configuration Model 63
4.1.2 Application Dynamic Model 66
4.2 Component Model (CM) 68
4.2.1 Component Interfaces 68
4.2.2 Component Properties 71
4.3 Machine Model 74
4.4 Failure Modeling 76
4.4.1 Cause of Failures 77
4.4.2 Types of Failures 77
4.4.3 Behavior of Failed Systems 78
4.4.4 Failures in Our Model 79
4.5 Constructing a Domain 82
4.5.1 Scope of the Domain 82
4.5.2 Predicates and Functions 83
4.5.3 Predicates 84
4.5.4 Functions 88
4.5.5 Actions 88
4.5.6 Actions and Interface 95

5 Automating the Failure Recovery Process 96
5.1 Failure Scenario 96
5.2 Sense 98
5.2.1 Failure Detection Models 98
5.2.2 Failure Detection in our Model 102
5.3 Introduction to AI Planning 106
5.3.1 Planning Inputs 106
5.3.2 Explicit and Implicit Configurations 108
5.3.3 Planning Output 108
5.4 Analyze 109
5.4.1 Determining Effects of a Failure 109
5.4.2 Gathering Complete Application Description 111
5.5 Plan 113
5.5.1 Initial and Goal States 113
5.5.2 Goal State 115
5.5.3 Plan 118
5.6 Execute 122

6 Failures During Failure Recovery 124
6.1 Handling Recovery Failures 124
6.1.1 Recovery Failures During ‘Plan’ Phase 125
6.1.2 Recovery Failures During ‘Execute’ Phase 127
6.2 Handling Failures during Failure Recovery 129
6.2.1 Kinds of Dependencies 129
6.2.2 Dependency State 131
6.3 Example System 132
6.4 Planning 132

6.4.1 Handling Failure During Planning . . . 134

6.5 Plan Execution . . . 136

6.5.1 Handling Failure During Plan Execution . . . 137

7 Implementation 141

7.1 Monitor . . . 141

7.2 Machine and Component Agents . . . 144

7.3 Current Configuration Inspector . . . 145

7.4 Models . . . 146

7.5 Dependency Analyzer . . . 146

7.6 Problem Manager . . . 147

7.6.1 Planner . . . 148

7.7 Plan Manager . . . 148

7.8 Execution Manager . . . 149

7.9 Execution Agent . . . 149

7.10 Target Configuration Manager . . . 149

8 Evaluation 151

8.1 Basic Experiments . . . 151

8.1.1 Experimental Setup . . . 156

8.1.2 Induced Failures . . . 156

8.1.3 Measurements . . . 157

8.1.4 Experimental Results . . . 160

8.2 Synthetic Experiments . . . 161

8.2.1 Induced Machine Failures . . . 165

8.2.2 Induced Component Failures . . . 166

8.2.3 Computational Complexity Issues . . . 166

8.3 Intensive Experiments . . . 168

8.4 Conclusion 172

9 Future Research and Conclusion 174

9.1 Future Research 174

9.1.1 Dealing with Failures during Failure Recovery 174

9.1.2 Addition of More Resources in the System 175

9.1.3 Developing a Framework for Developing Scripts for Reconfiguration 175

9.1.4 Automated Mechanism to Develop a Planning Domain 175

9.1.5 Dynamic Reconfiguration to Improve System Performance 175

9.1.6 Distributed Systems Simulator 176

9.1.7 Failure Forensics 177

9.2 Conclusion 177

Bibliography 178

Appendix

A Initial and Goal State 187

B Plan 191

C Failure Recovery Planning Domain 192

Tables

Table

6.1 A summary of what happens to the recovery process when more components fail. 140

8.1 Recovery time for application Rubbos after its failure on Skagen 159

8.2 Recovery time for application Rubbos after its failure on Serl 159

8.3 Recovery time for application Rubbos after its failure on Leone 159

8.4 Machine Failures: Time to find the First Plan 161

8.5 Machine Failures: Time to find the Best Plan 164

8.6 Component Failures: Time to find the First Plan 164

8.7 Component Failures: Time to find the Best Plan 164

8.8 Machine Failures: Corresponding Actions and Facts from the Planner LPG167

8.9 Component Failures: Corresponding Actions and Facts from the Planner LPG 167

8.10 Recovery Process After First Failure 170

8.11 Recovery Process After Second Failure 171

8.12 Recovery Process After Second Failure 171

Figures

Figure

1.1 A Redundant Architecture for Failure Recovery 3

2.1 Flow of Sekitei Planning System 31

2.2 Dependency Relationship Among Components 33

2.3 Dependency Classification by Keller and Kar 35

2.4 Related Work Classification 40

3.1 Various Configurations of Internet-Scale Systems 55

3.2 Architecture of System used in this Dissertation 58

4.1 Predicates used in the Domain 86

4.2 Predicates used in the Domain 87

4.3 Graphical Representation of the Actions for Goal Predicate *(application-ready-5 ? app ?ap ?s ?t ?con)* 93

5.1 Failure Flow Figure 97

5.2 System before, during and after Failure Recovery 99

5.3 Push Model (from Felber et al.) 100

5.4 Pull Model (from Felber et al.) 101

5.5 Dual Model (from Felber et al.) 102

5.6 Planning Domain, Initial State, Goal State and Plan 123

6.1 Further Failures Possible During Recovery 130

6.2 Failure detection that results in a decision to either continue or restart the recovery phase. 131

6.3 Dependency graph of system components. 133

7.1 A High-Level Architecture of “Recover” 142

8.1 Experimental Setup for Basic Experiments 152

8.2 Failure Recovery after Failure of Machine Skagen 153

8.3 Failure Recovery after Failure of Machine Serl 154

8.4 Failure Recovery after Failure of Machine Leone 155

8.5 Planning time to find a Plan after Machine Failures 162

8.6 Planning time to find a Plan after Component Failures 163

8.7 System State After two Consistent Failures 169

Chapter 1

Introduction

Failure recovery procedures are required to provide high availability in distributed systems. However, many existing failure recovery techniques have a considerable period of downtime associated with them. This downtime can cause a significant business impact in terms of opportunity loss, administrative loss and loss of ongoing business. There is a need not just to reduce the downtime in the failure recovery process but also to automate it to a significant degree in order to avoid errors that are caused by manual failure recovery techniques [79].

Failures in distributed systems can be unpredictable in that they can leave the system in one of many possible failed states. Further, there may be several different acceptable recovered states. These may range from configurations providing minimal functionality all the way to complete restoration of functionality. This combination of many failure states with many recovered states complicates recovery because it may be necessary to get from any failed state to any recovered state. Computing the recovery path may delay recovery and may cause the system to be down for a considerable period of time. To counter this problem a mechanism is required that can take into account the state of the system after a failure and to search for a way to bring the system back to a chosen working condition.

To further complicate matters, the failure of one component can have ripple effects on other parts of the system. Moreover, the ripple effect may not be obvious. One

component may fail and cause a dependent component to stop functioning without clear symptoms of failure.

The present failure recovery techniques in distributed systems build scripts to carry out the failure recovery process. These scripts are executed on systems to recover them after a failure. Writing these scripts is a highly manual process. Therefore, human operators have to spend a large amount of time in writing them. In some failure scenarios a script may not be readily available for failure recovery. Moreover, due to the large and increasing size of the component based system, manually writing an optimal script is difficult.

To ameliorate these problems, a few systems use a library of pre-written scripts to use in such situations. There are a few problems with this approach. The first problem is that, after a failure systems can go into innumerable states and writing scripts for each of these states is nearly impossible. The second problem is that the criteria for choosing the best script from the library of scripts can be difficult to obtain because there may be more than one script available for failure recovery in a given situation.

Other problems with failure recovery are the scenarios when a system fails in the middle of a recovery process. Here the recovery process can fail in two ways. First, when a failure occurs, the goal of the recovery process is to recover the system to the pre-failure configuration. However, it may be possible that due to unavailability or catastrophic failure of a component or machine the pre-failure configuration is not possible. Therefore, the recovery process must be intelligent enough to select a different configuration and apply that configuration to the system to recover it. Second, when a system is recovering from a failure, other previously working components or machines san also fail. Therefore, the failure recovery process has to take into account these further failures into account before progressing further. In the worst case the whole failure recovery process may need to be started again in this case. An automated failure recovery mechanism must take all these factors into account while recovering a system

from a failure.

1.1 Failure Recovery Techniques in Distributed Systems

A number of failure recovery techniques have been proposed for internet-scale distributed systems. However, most of these techniques are not used in large-scale systems due to a number of limitations. In this section we will discuss two techniques that are often used in large-scale systems.

These two techniques are 1) redundancy and 2) failure recovery scripts.

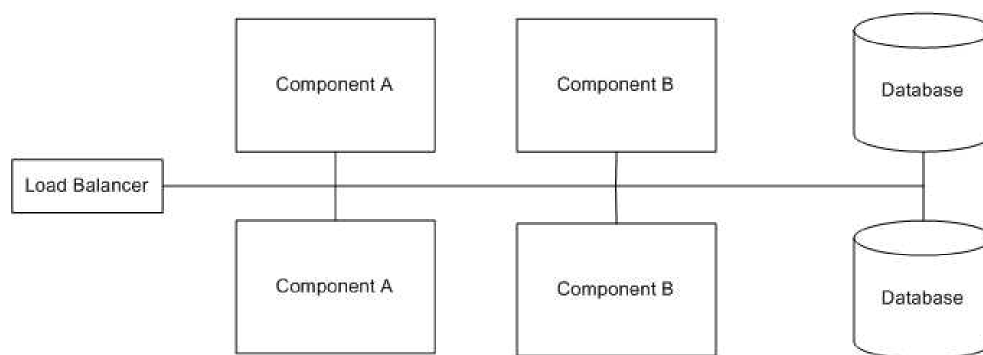


Figure 1.1: A Redundant Architecture for Failure Recovery

1.1.1 Redundancy-based Failure Recovery

In a redundancy-based technique, the system is augmented with more hardware. Two or more replica or clones of the system or parts of the system run at the same time. In case of a failure of one clone, the other clone takes over. Redundancy is achieved through a number of architectures. The best way, and indeed, the most costly way is to have a seperate component like Tomcat or Apache on a seperate piece of hardware. One such example is given in Figure 1.1 where each component in the system has a clone. In case of a failure of any component the identical component takes over. A load balancer is sometimes used to send the request to the working component.

As the cost of hardware keeps going down the redundancy based techniques are a common way to acheive high availability. However, cost of hardware is just one variable in the working of these systems. Other variables like maintenance cost etc. also need to be taken into account. Total cost of ownership (TCO) is a measure that tells the total amount needed to keep a system online. Typically the total cost of ownership is 5-10 times higher than the total cost of hardware and software. One third of the TCO is used to keep the system from failing or to recover the system after a failure. Therefore, although a redundancy based approach is a common solution to high availability, the costs associated with it are too high. Furthermore, downtime due to failure in redundant systems publicized as highly available ranges from 8 to 80 hours per year [79].

1.1.2 Failure Recovery Scripts

Failure recovery scripts are also used to recover the system after a failure. However, the current process of writing these scripts is highly manual. An administrator writes these scripts, when a failure occurs these scripts are run, in many cases, manually onto the system.

It is practically impossible to write a script for each possible failure for each component in the system [78]. Therefore, if a failure occurs and the script is not available then the administrator has to write the script manually and then execute it onto the system. Moreover, due to the change in the topology of the system following a failure, a particular configuration may not be achieveable. However, another configuration may work. Scripts are usually not able to recover the system in another configuration because each configuration of the system may have a different set of requirements and dependencies. Including all the requirements and dependencies may make the script too hard to write and maintain.

Both of the aforementioned techniques increase cost and/or time for failure recovery. Therefore, it is desired that a new technique be developed to automate the failure

recovery process such that the system is highly available.

The goal of this dissertation is to automate the failure recovery procedure in distributed systems. The approach to failure recovery used in this dissertation is based on Artificially Intelligent (AI) planning. As we will see in this dissertation, AI planning in combination with other techniques and tools can be a powerful mechanism for the failure recovery in distributed systems.

1.2 Approach to Automating Failure Recovery

The main idea of our approach is taken from the Sense-Plan-Act (SPA) mechanism in control systems. In control systems different system variables are maintained within specified bounds with a certain mechanism. The values of these variables are measured against a reference input. If the value of any variable is detected to be beyond established bounds a planning process is initiated. This planning process uses control algorithms to maintain the values of these variables within bounds. The results of the control algorithm are applied to the system to bring the variable within bounds.

The essence of this approach is used in dynamic reconfiguration because the goal of dynamic reconfiguration is also to maintain the system in a stable state [8, 63]. *Sensing* is the detection of a need for reconfiguration in a distributed system. The need for reconfiguration may be automatic or may be initiated by a user. Automatic initiation of reconfiguration may result from performance degradation or an outside attack, etc. Sensors and detectors are embedded in the system to provide the information about a need for dynamic reconfiguration. *Planning* is a search mechanism to find the steps needed to go from one configuration to another configuration. Planning could be performed through various techniques like reconfiguration languages, reconfiguration algorithms, pre-written scripts, AI planners etc. A plan is then executed on the system to bring the desired change. This phase is called *Acting*.

Failure recovery is also a reconfiguration of the system from a failed state to a

working state. However, the original SPA technique has been modified in this dissertation. Distributed systems are often made up of heterogeneous components that vary in size and behavior. Therefore, an analysis step is required to check the ripple effects of a failure to other components in the distributed system. Therefore, we have added another phase called Analyze in the original SPA technique. Moreover, since Act is more like executing we have changed its name to Execute to show a more software related approach. Hence, our model of failure recovery is Sense-Analyze-Plan-Execute (SAPE). This model is similar to IBM's autonomic computing model [56]. However, the difference is that IBM's model is a general model for self-managed systems, whereas our model is geared heavily towards failure recovery of distributed systems.

1.3 Planning and Failure Recovery

A cornerstone of our approach is artificially intelligent (AI) planning. AI Planning is useful in recovering from failures in distributed systems for a number of reasons. The first reason is the wide spectrum of failure scenarios. In these failure scenarios it is not practical to enumerate all possible types of failures in a large system. Moreover, there are a number of ways to recover each failure based on the target configuration of the system. Planning is used traditionally in situations where it is not possible to enumerate all the possibilities of moving from one state to another, and so planning should be helpful in recovering the system from a failure situation.

The second reason is the capability of planners to plan from an initial state to a goal state given the semantics of the system. In failure scenarios the initial state is the failure state – the actual current state after the failure. The goal state is the target configuration where the systems should be after recovery. The planning process constructs the sequence of steps needed for the system to move from a failed state to a target configuration state.

The third reason is the ability of planning to minimize time, cost and resource

usage. As stated before there are a number of ways to recover from a failed state. Each one has its own time, cost and resource constraints. By using planning one can obtain a near optimum plan by specifying priority matrices. True optimality, of course, may not be achieved because of limits on the amount of time given the planner to produce a plan.

1.4 Contributions and their Evaluations

This dissertation makes a number of contributions in automating the failure recovery in distributed systems. First, it makes it easy to specify higher level goals for carrying out failure recovery. Therefore, whenever a failure occurs in the system only higher level goals need to be specified and the automated failure recovery process takes care of the rest of the recovery process. In this way the administrator does not have to fiddle with low level configuration directives that are hidden inside the configuration files. Second, it automates the failure recovery in a number of failure scenarios and relieves the administrator from manually writing failure recovery scripts. Third, by using AI planning it optimizes the cost, time and resources. Moreover, by using planning the overall failure recovery time or Mean Time To Repair (MTTR) is reduced. Fourth, it recovers the system in the configuration that the system has before the failure. However, if for some reasons that configuration is not possible then the system is recovered in another configuration which is lighter in terms of resource requirements and dependencies.

To validate the aforementioned contributions we have carried out a number of experiments. Our experiments can be divided into three broad categories: basic experiments, synthetic experiments and intensive experiments. In *basic experiments* we have developed an end to end failure recovery system called 'Recover'. This recovery system is applied on real world applications developed by other colleagues. We induced failures of different kinds in those applications and measured the failure recovery time

that ‘Recover’ took in recovering those applications. After basic experiments we found out that the bottleneck of the recovery process is the planning time required by the AI planner. Therefore, we conducted *synthetic experiments* with a number of applications to test the planner’s capability to plan failure recovery in large-scale internet systems. The third type of experiments are *intensive experiments* which test ‘Recover’ to its limits. In these experiments we induced failure after failure in the system and measured the capability of ‘Recover’ to recover from a series of failures. Another goal of these experiments is to test ‘Recover’ to see its capability in recovering the system to a relatively light configuration if the original configuration is not possible.

The results of these experiments are very encouraging. Our failure recovery system ‘Recover’ is able to recover systems from failure in a very short period of time in the basic type of experiments. In most of these experiments the recovery time has been less than 15 seconds. In the synthetic experiments we calculated the planning time because it seems to be the most contributing factor in overall recovery time. In these experiments, the planning time for a system containing up to twenty machines is approximately 80 seconds or less. Moreover, the intensive experiments also provided good results in situations where extended failures are reported in the system. Details of these experiments and their results are discussed in chapter 8.

1.5 Scope and Assumptions

In this dissertation the focus is to recover a system at the application level. This dissertation does not deal with operating system level or firmware level recovery. We use off the shelf widely available components to show the promise of this technique. Moreover, we will be recovering stateless applications and components. Recovering of stateful components is out of scope of this dissertation. Our model involves three major artifacts, i.e. machine, component and application. An application is deployed on the component and a component is deployed on the machine. All three of them have a

fail-stop behavior; that is, they stop working immediately after a failure. Therefore, no erroneous output is expected from them following a failure. Our assumption is that the network is reliable and messages and events like failure notifications are delivered within specified bounds. Moreover, no false positives or false negatives are possible. Moreover, we are not taking into account database related failures in this dissertation. Databases already have good built-in failure recovery procedures. Therefore, we only deal with components that are higher level than databases. Also, the failure recovery system itself does not fail. Moreover, we assume that our system does not intrduce worse errors. Furthermore, dependencies and configurations of various applications in the system are correctly specified.

1.6 Organization of the Dissertation

In chapter 2 we give an overview of foundational and related work. Foundational work are tools, techniques and frameworks that we have used as the basis of this dissertation. In related work we give other approaches which compete or complement the work in this dissertation.

In chapter 3 we give a motivating example of a real world scenario. In this real world scenario we discussed an internet-based system which requires automated failure recovery. We also give details of the applications that we are using to evaluate our failure recovery technique. These applications are not developed by us but by other colleagues.

Chapter 4 gives a detailed overview of the modeling required to automate failure recovery in distributed systems. We discussed a number of models that make it possible to view a complete system state and to decide about failure recovery. Moreover, we discussed the types of failure being modeled in this dissertation. Furthermore, we discussed the development of a planning domain. A planning domain encapsulates the semantics of failure recovery. Therefore, developing a good planning domain contributes

a lot to automated failure recovery.

Chapter 5 gives an overview of the automation of failure recovery process. In this chapter we discuss the four phases of failure recovery which include sense, analyze, plan and execute. All these phases are discussed with their contributions to the overall failure recovery process.

Chapter 6 discusses two exceptions that occur during failure recovery process and how to deal with those exceptions. The first exception is the inability to restore the original configuration of a system following a failure. Second, is the handling capacity of further failures in other parts of the system when the system is already recovering from failure.

Chapter 7 gives an overview of the implementation. In this chapter we discuss the architecture of our failure recovery system called 'Recover'. We discuss all the required and optional modules of 'Recover' and their role in the recovery process.

Chapter 8 presents our evaluation results. Three types of experiments are carried out to evaluate our contributions: basic experiments, synthetic experiments and intensive experiments. These evaluation results are measured after many test runs on the target system.

Chapter 9 gives a conclusion and some interesting further directions that stem from this dissertation.

Chapter 2

Foundational and Related Work

Failure recovery is based upon a number of foundational approaches. Moreover, many related approaches to failure recovery are in place. Therefore, we divide this chapter into two major sections: foundational work and related work.

2.1 Foundational Work

The foundational work of failure recovery is based on many areas in computer science, such as fault tolerant computing, dynamic reconfiguration, dependency analysis and so on. We present some foundational work from each area in this section.

2.1.1 Dynamic Reconfiguration

The purpose of dynamic reconfiguration is to allow a system to evolve incrementally from its current configuration to another configuration without being taken offline [7]. Various approaches to dynamic reconfiguration have been proposed in the literature. In this section we present an overview of some of those approaches.

2.1.1.1 Agent-Based Approaches

In agent-based approaches an agent or a set of agents are responsible to carry out the dynamic reconfiguration. These agents are mobile or stationary. Valetto et al. proposed a mobile agent-based approach to carry out dynamic reconfiguration [98]. They

claim that process-based dynamic reconfiguration solves the problem of maintenance of the software system at a less cost than reengineering. Their agents or mobile code are called worklets. These worklets carry code in the system and carry out the sequence of reconfiguration. However when the system becomes complex pre-built scripts are required in these agents. They have developed a system called KX which is a meta architecture on top of the target system. Such meta architecture is in charge to introduce an adaptation feedback and feedforward control loop onto the target system, detecting and responding to the occurrence of certain conditions. The worklets then carry out the dynamic reconfiguration when it is required.

Another approach for implementation of dynamic reconfiguration using mobile agents is proposed by Berghoff et al. [14]. According to them both the central and the decentralized schemes for dynamic reconfiguration have their disadvantages. The disadvantages of centralized system are the constraints on scalability and resources; violation of which can cause network clogging. In a decentralized management system the problem of network clogging is solved but there is more traffic due to synchronization and coordination. Therefore, the authors have proposed an approach based on mobile agents. These mobile agents can move in the system with their state. These agents perform the management task in isolation. Thus, there is no network clogging due to extra bandwidth usage. Using mobile agents, management tasks can be carried out simultaneously. The micromanagement is performed locally which reduces network traffic. Mobile agents collect information about the system state and send it back to the management node. According to the authors, the aim of this research is not to provide an entirely new management framework but to extend the present centralized management approach and add agents to it. The authors have described an agent-based extension to the PRISMA management environment. Mobile agents are safe because they can work in a controlled environment. Agents have an agent server and they communicate through an information space.

Castaldi et al. proposed a light weight infrastructure for the reconfiguration of applications called LIRA [23]. LIRA is a light weight infrastructure for managing dynamic reconfiguration that extends the concepts of network management to component-based, distributed software systems. There are two types of reconfiguration the authors have mentioned.

- Internal Reconfigurations, which rely on programmers. These reconfigurations are built into the component by a programmer.
- External Reconfigurations, which rely on some external entity. This external entity decides when and how to perform the reconfigurations.

The authors have defined a lightweight approach to reconfiguration. According to this definition it is a service which uses facilities already provided by the component. Moreover, reconfiguration is carried out remotely, in that the management of reconfiguration is separated from the implementation of reconfiguration. Furthermore, communication is through a simple protocol between components and entities managing their reconfiguration, rather than through complex interfaces and/or data models.

This approach uses a network management model. A network management model has four elements: agents, protocols, managers and management information. These agents are not mobile agents but instead are interfaces to the information contained in an application or a subset of application on a host.

Chen presented a framework of component development with built-in component reconfiguration [27]. This framework has a CM agent for managing the dynamic reconfiguration in the system. The authors claim that the present frameworks of components like CCM, EJB and DCOM does not provide much support for dynamic reconfiguration. Also they say that the present frameworks either put a lot of work on programmers or they put a lot of restrictions in developing component-based applications. Therefore, they have developed a method for determining dependencies in the components at run

time. They have classified dependencies into two categories: static dependency and dynamic dependency. The component framework offers dependence management that analyzes the dynamic dependencies between components. Moreover, it uses virtual stubs that not only realize location transparent invocations among components but also dynamically monitor and manipulate interactions among components. In addition the CM agent can automatically update an invalid reference to a component after its reconfiguration. In the component framework a consistent configuration is guaranteed and the disruption of the system is minimized. The reconfiguration is carried out at the framework level so the developer have to do minimal work while developing a component.

2.1.1.2 Redundancy-Based Approaches

In a redundancy-based approach the system has more than one version of components working at the same time. Different versions provide similar functionality at the same time. A goal of these approaches is to save the system from breaking because of inconsistencies developed due to new versions of the components.

Feiler and Li presented a technique called Analytically Redundant Component (ARC) that provides a protection from application faults [40]. This paper is actually about dynamic reconfiguration in control systems. However, the authors have made some nice observations that can be used in any system that requires dynamic reconfiguration. They presented a technique which performs an offline analysis. This offline analysis determines inconsistencies in configurations and identify reconfiguration paths to recover the system to a consistent configuration. The results of this offline analysis are used by a configuration manager to avoid configurations that are inconsistent. They have used the simplex fault tolerance capability and augmented it with proactive fault avoidance through detection and recovery from inconsistent reconfigurations. Before deciding any configuration the configuration manager performs a few sanity checks. If

a given configuration inconsistent, it should be avoided as a target configuration. The configuration manager checks the system for questions such as, is there a desirable configuration for an observed fault or inconsistancy? and What is the impact of the change being carried out? and how this impact can be reduced?

A given configuration is considered wellformed, if it is complete and consistent. Complete means that all component ports are connected and all connectors have a source and destination. Consistent means that connectors are attached to component ports and the direction of the connection matches the direction of ports.

Cook and Dage also presented a redundancy-based approach in their system HERCULES [30]. They argue that when a user is comfortable with a version of component then it is difficult for him/her to adapt another new version of the component. Therefore, they presented a technique in which the old and new versions of the component work at the same time. When the user is comfortable with the new version of the component then the old component version can be replaced permanently. Moreover, they say that instead of changing the whole system, components should be replaced one by one.

The authors have developed a framework called HERCULES. In this framework they have taken the approach that has the following features. The framework formally specify a specific sub domain that a new version of the component addresses. Multiple version of the components are allowed to run in parallel. The results of multiple versions of the same component are selected from the version whose specified domain contains this invocations parameters. In case of ambiguity, a voting mechanism is used. All the data is logged to check the correctness of all versions of a component.

2.1.1.3 Operating System (OS) Level Approaches

OS level approaches to dynamic reconfiguration use either components at the operating system level or facilities at this level to manipulate the system using OS level

commands. Soules et al. presented a technique for dynamic reconfiguration of the operating system level components [92]. There are two mechanisms of online reconfiguration provided by the authors. interposition and hot swapping. In order to implement hot swapping and interposition, four capabilities are required: First, systems must be able to identify and encapsulate the code and data for a swappable component. Second, the system must be able to make the swappable component quiescent. Third, the system must be able to transfer the internal state of a swappable component to the replacement component. Finally, the system must be able to modify all external references to a swappable component. Out of these four capablities, all of them are required for hot swapping but only the the fourth one is required for interposition.

Interposition is a technique that wraps an active components interface. Thus, extends its functionality. Interposition wrappers can be for a specific component or they can be generic and can be used for a variety of components.

Hotswapping replaces an active component with a new component instance that provides the same interface and functionality. To maintain availability and correctness of the service, the new component picks up from the state where the old one left off.

The authors have also identified a number of areas where these two techniques can be used. For example, patches and updates, adaptive algorithms, dynamic monitoring and so on.

Hauptmann and Wasel presented a technique for dynamic reconfiguration that uses the operating system level services [51]. The authors gave an on the fly software replacement technique mainly for PABXs and other embedded software systems. They define on-the-fly software replacement as maintenance in a running system with minimal interruption of service.

The authors provided a reconfiguration approach, which is dependent heavily on the underlying operating system 'Chorus'. The technique they describe is soft real time, which means that a timing failure does not cause a catastrophic failure in the system.

A replaceable unit in Chorus is an actor. An actor is defined as a unit of resource allocation (similar to a heavy weight process). An actor has a protected address space. It manages its own ports and it contains any number of threads. An actor is a unit of configuration. It has internal and clearly defined external linkages. If more than one actor is involved in reconfiguration then the actors are replaced in a sequential manner. It helps to keep the soft real-time requirement, easier to control the replacement process and memory needs of the program are smaller.

The algorithm to replace a single actor works as follows:

- Load the new actor in the address space.
- Stop all application threads of the old actor.
- Collect the state of all objects of the old actor.
- Transmit these states to the new actor.
- Migrate all ports of the old actor to the new actor along with all pending messages.
- Map object structures between the old and the new actor.
- Start the threads of the new actor.
- Delete the old actor from the memory.

2.1.1.4 Platform and Application-Based Techniques

Different techniques are proposed for dynamic reconfiguration that are based on specific platforms or applications like EJB, CORBA, etc.

Batista and Rodriguez presented a technique for dynamic reconfiguration in CORBA [13]. They use configuration languages for dynamic reconfiguration. According to them a

component has two parts i.e. an interface part and an implementation part. Configuration can only be performed at the interface part. The authors use a configuration language called 'Lua' for the dynamic reconfiguration. Lua has different parts like LuaSpace and LuaORB. According to the authors Lua has the capability to support both programmed and ad-hoc reconfigurations. The authors argue that their system support reconfiguration at two levels. In the first one a programmer defines components that offer the required services and in the second one the programmers declare only services that compose an application and generic connectors select automatically the proper components to execute them.

Almeida et al. presented a dynamic reconfiguration technique for CORBA [7]. The authors argue that granularity of reconfiguration is important before deciding to implement dynamic reconfiguration in a system. A reconfiguration design activity is required before a reconfiguration takes place. This activity is a kind of a plan where specification and constraints are specified. During the reconfiguration a system must preserve consistency. In order to hold correct a system must keep three properties: First,the system satisfies its structural integrity requirements. Second, the entities in the system are in mutually consistent states and third, the application state invariants must hold. The reconfiguration must be performed in a reconfiguration safe state. The authors then discuss dynamic reconfiguration service requirements. They give a dynamic reconfiguration for CORBA middleware using their approach.

Rutherford et al. presented a reconfiguration tool called BARK for reconfiguration in the Enterprise Java Beans Component Model [87]. Little and Wheather presented a technique for dynamic reconfiguration in applications that are built in Java [67]. The authors claim that the reconfiguration is required in Java applications because of the different security accesses for different users. By providing dynamic reconfiguration, the applications dynamically adapt themselves to the types of security restrictions that exists when they are executed. They also say that in web applications dynamic re-

configuration is used to provide adaptive security. The authors have used the Gandiva model, which have a separate notion of an interface component and an implementation component.

2.1.1.5 Middleware-based Approaches

In middleware-based approaches, a middleware is responsible for the dynamic reconfiguration of the system. Palma et al. presented a middleware-based approach for dynamic reconfiguration of agent-based applications [76]. The authors give two goals for their work. The first one is to develop a general approach for dynamic reconfiguration and the second one is to build system level tools to implement it.

They say that the dynamic reconfiguration covers four issues: modifying the architecture of an application, modifying the geographical distribution of a an application, modifying agent's implementation and modifying agent's interfaces.

The dynamic reconfiguration approach that they proposed is for an agent-based system specified in an ADL. There are three operations that can be performed on agents:

- Rebind: where change of a reference held by an agent for another agent.
- Move: allows the agent to migrate to another site.
- Delete: allows an agent to be removed from an application.

Furthermore, there are also three states in which an agent can go. Active, which is normal execution state. Passive, where agents can react to events but cannot send events and Frozen, where agents cannot receive any more events.

The authors give an algorithm for carrying out the dynamic reconfiguration:

- Compute the agent passive state(APS).
- Passivate all agents of APS.

- Send the reconfiguration order to target agent.

The authors have applied this approach to a message oriented middleware (MOM).

Paulo et al. presented a middleware-based approach for the dynamic reconfiguration in CORBA components [80]. They give the requirement of a dynamic reconfiguration approach which includes correct incremental evolution, general applicability, scalability, impact on execution, responsibilities and transparencies and finally configuration information.

They provide the structural integrity in CORBA by providing referential integrity and interface compatibility. They provide the mutual consistency in CORBA by driving the system into a safe state and then applying the reconfiguration. The application invariants hold by a good reconfiguration design.

2.1.1.6 Workflow-based approaches

In workflow-based approaches the dynamic reconfiguration is carried out as task network. Two such approaches have been presented in the literature. Kaiser et al. uses a workflow engine called Workkflakes (a decentralized workflow engine) for the dynamic reconfiguration of the component-based systems [54]. This system helps to carry out local adaptations and more global reconfigurations. Workflakes coordinates the actual reconfiguration by invoking low-level effectors. Worklets (kind of mobile agents) are the effectors in this case. Workflakes provide a reconfiguration workflow by selecting, instantiating and dispatching worklets and coordinate the activates of the Worklets.

The second such approach is presented by Shrivastava and Wheather [91]. They designed an application composition and execution environment and executed a transactional workflow system that enables sets of interrelated tasks (applications) to be carried out and supervised in a dependable manner. A task model that is expressive enough to represent temporal dependencies between constituent tasks has been

developed. Transactions for dynamic reconfiguration ensures that the transactions are carried out atomically. The authors have developed a transactional workflow system that enables sets of inter-related tasks to be carried out and supervised in a dependable manner.

2.1.2 Planning

Another area that provides a foundational approach to automated failure recovery is planning. Planning is used to develop a way to change the state or configuration of the system. Two issues are of interest here: First, how to represent a system such that planning can be performed on it and second, the planning process itself that takes the system from one state to other.

2.1.2.1 Models and Frameworks

Frameworks and models are one of the ways to perform planning for dynamic reconfiguration. A model is defined as an abstraction to capture the structural and runtime properties of the system. On the other hand a framework is defined as a guideline for developing an application such that dynamic reconfiguration is possible in the application. Chen et al. [27] gave two requirements for frameworks that provide dynamic reconfiguration. The first requirement is that the framework must have the knowledge of interactions going on in the system. In the case of reconfiguration the framework can block interactions so that the components reach a reconfigurable state. The second requirement is that the framework must provide location transparency.

Whisnant et al. presented a bottom up system model for reconfiguration [105]. In this reconfiguration model the system is represented by a triple (C, V, T). C is the set of code blocks in the system. Code blocks perform a computation when triggered by operations. V is a set of state variables in the system. T is the set of threads in the system. In this model the components are the subsets of state variables, called elements.

The reconfiguration is performed on elements and threads. Each code block has an associated signature. These signatures collectively are used to analyze the dependencies in the system.

A second system model is presented by Hauptmann et al. [51]. In this system model a node or site is a set of tightly coupled resources. On each node there is an arbitrary number of actors. The replaceable unit in this case is an actor. An actor is defined as a unit of resource allocation (similar to a heavy weight process). An actor has a protected address space. It manages its own ports and it contains any number of threads. An actor is a unit of configuration. It has internal and clearly defined external linkages.

After discussing two system models we are now going to discuss frameworks that supports dynamic reconfiguration. One such framework is given by Chen and Simons [27]. They claim that the present frameworks of components like CCM, EJB and DCOM does not provide much support for dynamic reconfiguration. Moreover, the frameworks put a lot of work on programmers or they put a lot of restrictions in developing the component-based applications.

They have developed a method for determining dependencies in the components at run time. They have classified dependencies into two categories: static dependencies and dynamic dependencies.

The component framework offers dependence management that analyzes the dynamic dependencies among components, and used virtual stubs that not only realize location transparent invocations among components but also dynamically monitor and manipulate interactions among components during a dynamic reconfiguration.

Hall et al. presented a framework called Software Dock [50]. This is an agent-based component deployment framework. This framework is basically geared towards the software producer and consumer relationship. The software producer supports processes like release and retire. On the other hand the software consumer supports pro-

cesses like install, activate, deactivate, reconfigure, update, adapt and remove. The Deployable Software Description (DSD) provides a schema for describing a software family. A server is residing at each consumer site that has the model of the system in terms of the configuration and the set of system resources. This server serves as an interface to the consumer site for any consumer related processes.

2.1.2.2 Dynamic Reconfiguration Languages

A dynamic reconfiguration languages is a system-independent way of representing a system so that the reconfiguration can be planned and executed on it. Various approaches have been suggested in literature. One of the first such approach was suggested by Kramer and Magee [63]. They argued that the changes must be applied at the structural level as opposed to the application level. They described a simple system for change management. This system consists of a system (set of processing nodes), node (a processing entity) and nonnection (a directed communication path from an initiator to the recipient, very close to a connector). A transaction is an exchange of information between two and only two nodes. A change is the modification to the system structure. Four actions are specified that act on the system: create, remove, link and unlink. The quiescence of a node is required to make a change. Quiescence is the property where the node is in a state in which the node is passive and there is no outstanding transaction which it must accept. The authors have described a change management protocol that has preconditions and the effects are the resultant state of the system. Also they say that the algorithm in this protocol is able to generate ordered set of actions and these actions can be done in parallel.

Endler and Wai presented a reconfiguration language called Gerel [69]. In their work the authors presented two types of dynamic reconfiguration. The first type is ad-hoc reconfiguration, which are performed when the system is already running and are performed interactively through a reconfiguration manager. The second type is

programmed reconfiguration which are preplanned changes specified at design time. These changes are used for fault recovery, automatic dynamic load balancing etc. The authors argue that some systems like Conic, Polylith and LADY only perform ad-hoc reconfiguration while some systems like Durra, Darwin and PRONET only support programmed reconfigurations. The authors argue that their solution combine these two types of reconfiguration using a language called Gerel.

The authors claim that configuration-based approaches for dynamic reconfiguration are the most successful, when a distributed system is built at two levels i.e. a programming level and a configuration level. At the programming level the components are built and at the configuration level they interact among each other.

The reconfiguration is performed using a change script written in Gerel. The script has two sections: precondition and execute. The precondition is used for reliability purposes, so that the change is only made when the particular state of the system is reached. This helps to perform an ad-hoc and a programmed reconfiguration at the same time. A sub language of Gerel called Gerel-SL is used to describe the structural properties of objects. It is also used to specify preconditions. Due to the power of the first order logic Gerel can express generic changes (changes that can be applied to a wide range of configurations).

Agnew et al. presented a language called Clipper for representing plans for dynamic reconfiguration [2]. A notion of reconfiguration plans is presented by them. A plan directs how the running system must be changed based on the events it receives from itself or the surrounding environment. Their language 'clipper' allows programmers to express plans that are used to generate the change actions. According to their definition the reconfiguration is a dynamic method in which the application is mapped between two execution states. The mappings may involve changes to the structure of the application (such as addition and deletion of modules or bindings) or it may involve altering the structure (called a configuration) with respect to host resources. Mappings

are expressed as plans in terms of a reconfiguration scripts. Clipper has a rule-based notation but still heavily dependent on C++.

The authors have also provided some requirements for reconfiguration languages which include their similarity with other high-level languages, external change control script, expressing the constraints in the language, reliability, performance and carrying out the reconfiguration that should be not noticeable for users.

To perform a dynamic reconfiguration, there are three possible stimuli for reconfiguration: the application, the distributed environment or the system environment. A reconfiguration module provides all the intended reconfiguration activities at run time.

Batista and Rodriguez used a configuration language called Lua [13]. They defined components having two parts i.e. an interface and an implementation part. Configuration can only be performed at the interface part. They used Lua for the dynamic reconfiguration of the systems. Lua has different parts like LuaSpace and LuaORB. According to the authors Lua has the capability to support both programmed and ad-hoc reconfigurations. The authors argue that their system support reconfiguration at two levels. At the first level, programmers define the components that offer the required services. At the second level, programmers declare only the services that compose an application and the generic connectors selects automatically the proper components to execute them.

Welch gave the idea of compensating reconfiguration [101]. He said that a running distributed application can be changed in three ways: structure, topography or implementation. Changing the structure involves adding and removing of components. In compensating reconfiguration an external condition is detected, an appropriate response is determined and if required the dynamic reconfiguration is carried out. The compensating reconfiguration must base its decision on the system reconfiguration, virtual world state and the mapping among them. The author used a system called Bullpen. Bullpen has three independently executing subsystems. ‘Scout’ detects the external

condition and reports any preliminary information to the Coach. ‘Coach’ contains the decision-making logic for both the compensation and reconfiguration decision. Coach has a knowledge base. Coach also executes the reconfiguration. Finally a ‘Scoreboard’ is responsible for keeping a copy of the state of the virtual and system state and their mapping.

Papadopoulos and Arbab used coordination languages for expressing dynamic reconfigurations [77]. They discuss the similarities among the coordination languages and the requirements for dynamic reconfiguration. The authors discuss a case study using a language called ‘Manifold’. An important claim that the authors made was that the quiescent state requirement is not required in case of using coordination language in dynamic reconfiguration.

2.1.2.3 Extensions to ADLs for Dynamic Reconfiguration

Architectural Description Language (ADLs) are used to represent the software architecture of a component-based system. However, the traditional ADLs lack in their ability to dynamically change the structure of the system. Various approaches have been suggested to add dynamic reconfiguration ability to existing ADLs. In this section we are going to review some of these approaches.

Aguirre and Maibaum presented a temporal logic approach for extending the ADLs for dynamic reconfiguration [3]. They argue that the present ADLs provide constructs for modeling the architecture of the system but they do not provide any mechanism for reasoning the evolution of the architecture. They say that according to their knowledge the specification languages for reconfigurable systems are either informal or they require the specification about evolution in an external meta language. The temporal logic approach have a notion of classes (components), associations (connectors and a new notion of subsystem for giving some modularity in component-based systems for reconfiguration. Axioms give the semantics to these artifacts. Each artifact has

its own set of axioms. This approach allows declarative specification and reasoning of component-based systems. It can express the properties of the architecture in a declarative way. The language is actually logic-based on a combination of first order logic and temporal logic. A proof calculi can be used to prove properties of the system including dynamic reconfiguration properties.

Wermelinger also extended the dynamic reconfiguration capability of the ADLs [104]. According to him the current set of ADLs have shortcomings when it comes to dynamic reconfiguration. First, the arbitrary reconfigurations are not allowed. Second, the level of abstraction does not match with the present programming techniques thus resulting in cumbersome specification. Third, the mismatch between reconfiguration and computation leads to additional formal constructs. In this approach they claim to make the following contributions to the above shortcomings. Architecture, reconfigurations and connectors are represented and manipulated in form of a graph that has strong mathematical basis. The program design language is at a high-level of abstraction. Computations and reconfigurations are kept separate but related through colimit (idea of gluing together mathematical objects). Graphs provide an easy way of modification. Using these advantages there are several practical problems that can be handled like state transfer, removing a component in a quiescent state etc. The authors have tried to use the formality of ADL combined with the power of reconfiguration programming languages like Gerel and used algebraic specification for their work. Scripts define the reconfiguration. Humans write these scripts. The scripts have commands and the order in which they are executed.

Wermelinger also suggested an extension to ADLs using the chemical abstract machine CHAM [102]. The chemical reaction model views computation as a sequence of reactions between data elements called molecules. The system is described by a multiset of molecules, the solution. The possible reactions are given by transformation rules: if the current solution contains molecules given on the left hand side of a rule, the rule

may be applied , replacing those molecules by the ones on the right hand side. There is no control mechanism, at each time several rules may be applied, and the CHAM chooses one of them nondeterministically. The solution thus evolves by rewriting steps. If no further rules can be applied, the solution becomes inert, i.e. stable.

Wermelinger and Fiadeiro presented a algebraic software architecture reconfiguration approach [103]. They suggested a uniform algebraic framework and a program design language with explicit state. The algebraic framework allows to represent both architecture and their reconfigurations and to explicitly relate the architectural level with computational level. The program design language provides some of the usual programming constructs while keeping a simple syntax. A reconfiguration rule is simply a graph production where the left hand side is the interface and the right hand side is the architectures. A reconfiguration step is a direct derivation from a given architecture G on an architecture H. Dynamic reconfiguration is a rewriting process over graphs labeled with program instances (i.e. architecture instances) instead of just the programs. This ensures that the state of components and connectors that are not affected by a rule does not change, because labels are preserved thus keeping the reconfiguration and computation separate. There can also be conditional rules that ensure the state preservation so the reconfiguration is carried out at a quiescent time. With the use of category theory, software architectures and their reconfiguration are both represented in graphical yet mathematically rigorous way. Computation and reconfigurations are related in a simple way through the colimit construction.

2.1.2.4 AI Planning

AI planning is another way to perform reconfiguration. Planning can be viewed as a type of problem solving in which the agent uses beliefs about the actions and their consequences to search for a solution over the most abstract space of plans, rather than over a space of situations [86].

Planning has been an area of significant research from the mid 1950s. Newell, Shaw and Simon developed the first system to solve planning problems. This systems name was General Problem Solver (GPS) [73]. After that many people have performed research in many areas of planning and performing the research from many perspectives.

A full detail of planning research is out of scope of this dissertation[1] . Instead, this dissertation will focus mainly on the planning problems that are related to the problem of dynamic reconfiguration of software systems. There are two ways in which we can see the related work in planning: Stand-alone planners and the systems that use these planners to provide enhanced planning capability like replanning.

The first perspective is stand-alone planners. These planners have been developed to solve a range of problems in many different areas. Here our focus will be on planning systems that take into account time and resource constraints for solving planning problems. There have been many research efforts that deal with temporal and resource planning [100, 5, 6, 81, 71, 47, 57, 58]. These and other approaches attack the temporal planning problem through various ways. Some of these approaches include graphplan extensions, model checking techniques, hierarchical decomposition, and heuristic strategies and reasoning about temporal networks. These approaches are capable of planning with durative actions, temporally extended goals, temporal windows and other features of time-critical planning domains.

Stand alone planners use various heuristics for searching. Kichkaylo et al. divided the heuristics of stand alone planners into four categories [58]: regression planners that start from a goal and move backwards to the start, progression planners that start from an initial state and move towards a goal, causal-link planner that perform means-end analysis and compilation-based planners that reduces the planning problem into a satisfiability problem.

[1] A detailed overview of planning is given in a planning textbook by Malik Ghallab, Dana Nau and Paolo Traverso entitled “Automated Planning: Theory and Practice” published by Morgan Kaufmann, 2004

In almost all of the research there has not been an agreement on the representation formats for representing the semantics of the planning problems. AI planners have not been standardized on one representation language up till very recently. Almost every planner needs a different set of input with varying format. Therefore it is very difficult to compare the effectiveness of AI planners in solving real problems. However, AI planning competition IPC 2002[2] provides a standard language PDDL 2.2 (Planning Domain Definition Language) that is used by many planners. This language can be used to define the semantics of the planning activity using a single syntax.

The usage of AI planning systems for solving real world problems has significantly increased in recent years. The European Network of Excellence in AI Planning (PLANET) [82] identifies key areas where planning can be applied. These areas range from Robot Planning to Intelligent Manufacturing. PLANET has identified the various strengths and shortcomings of the AI planners. They have proposed areas of improvement for further research in AI planning.

The second perspective in related work in planning is the planning systems that use these planners. There are various planning systems like Sipe-2 [42], ASPEN/CASPER [29], PRODIGY [99], etc. These planning systems usually use their own planner and have integrated the planner with an execution system. Most of these are built for specific applications and cannot be used as a general purpose planning system. For example ASPEN/CASPER has been developed for planning, scheduling and execution in space mission operations. Therefore, it has special features built into it for spacecrafts which are not necessarily useful in other domains.

Planning is one of the cornerstones of our approach, although no planners have been built for specifically handling the dynamic reconfiguration in component-based systems, recently there has been some work in making planners for component deployment [57, 58]. Kickkaylo et al. developed a planner called Sekitei specifically for

[2] http://planning.cis.strath.ac.uk/competition/

deployment of components in a resource constraint environment.

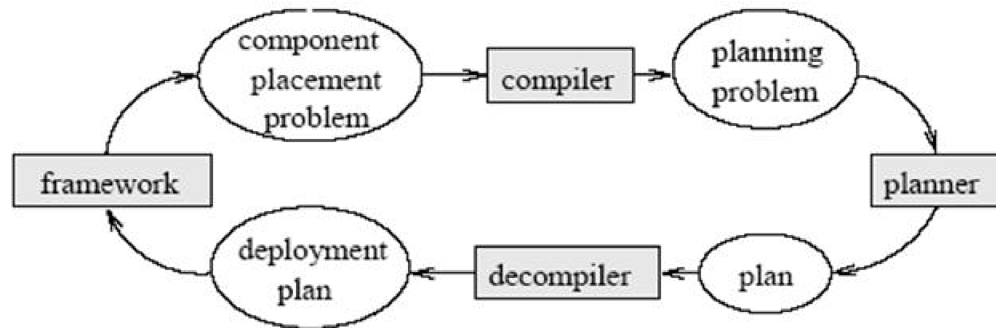


Figure 2.1: Flow of Sekitei Planning System

Sekitei is developed to solve the Component Placement Problem (CPP). The process followed by this planner for solving the CPP problem is shown in figure 2.1 [3] . In this process the planner works in conjuction with a component framework. The component framework dictates a component placement problem in XML. This CPP problem is translated into a planning problem using a compiler. The planner solves the planning problem and gives out a plan. The plan is the decompiled into a deployment plan and fed back into the component framework. The component framework then use this deployment plan to deploy the actual components in the network.

Sekitei planner is built to support the deployment process for use specifically in the grid-based and networked applications. It is able to handle resource optimization but it cannot handle temporal constraints.

2.1.3 Dependency Management

Component-based distributed systems have myriad depndencies at different level. In order to run such systems these dependencies must be fulfilled. Moreover, to automate the failure recovery these dependencies must be represented such that an automated failure recovery procedure is able to fulfill these dependencies.

[3] Adapted from Kichkaylo et al.

Various kinds of dependencies are present in distributed systems. Managing these dependencies is an important aspect of managing component-based applications. It is needed to get to the root causes of the problems in component-based systems. Moreover, it also helps in specifying a system structure from a functional point of view. There are various approaches for characterization of dependency management.

An automated system must take into accout all of these dependencies. However, many approaches to dependency representation are available. In this section we are going to look at three of these approaches.

Kon and Campbell specified two kinds of component dependencies [61]. These dependencies are dependencies that are required to add a component to the running system called prerequisite and the dynamic dependencies that are present at runtime.

In the perquisite there are three kind of dependencies

(1) the nature of hardware resources a component needs,

(2) the capacity of the hardware resources, and

(3) the software services such as, various components it require.

In the dynamic dependencies each component has a set of hooks to which the other component can attach. These hooks create a unidirectional dependency. The component that is attached to the hook of other component is called a client. An example of this relationship is given in figure 2.2[4] .

Whenever a component C depends on another component D the system must do two actions 1) attach D to the hooks of C and 2) add C to the clients of D.

Although the work by Kon and Campbell provided some good initial dependency relationships among components, there are other models of dependency management that are more enhanced and cover other kinds of dependencies also.

[4] from Kon and Campbell

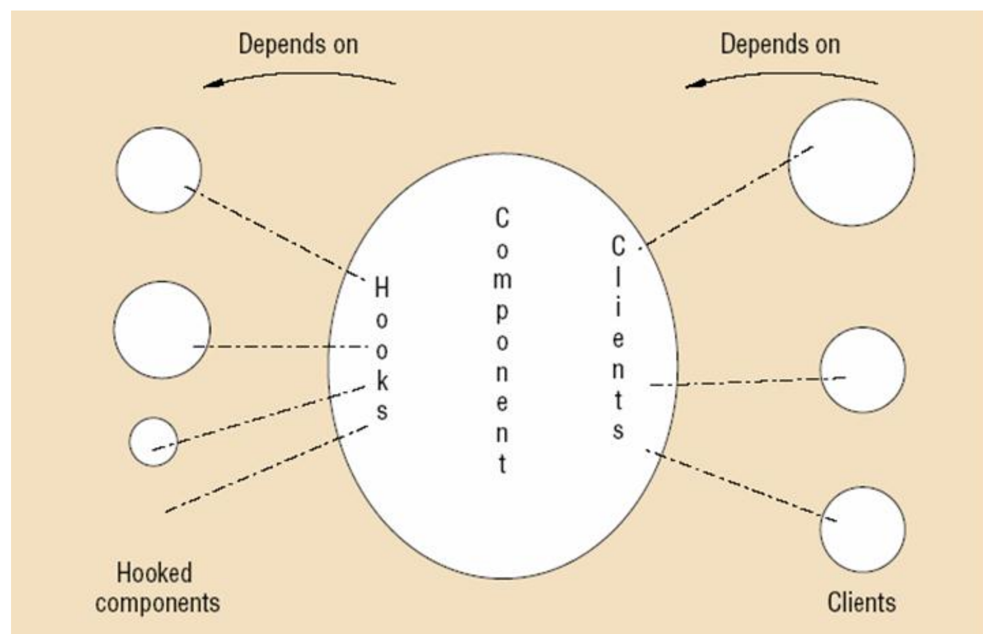


Figure 2.2: Dependency Relationship Among Components

One such effort was by Alda et al. They have given two groups of dependency classification. The first group are syntactical dependencies where the communication among the components actually takes place. The second group is also based on syntactical level but without direct communication among the components.

In the first group there are three kinds of dependencies a) Event Flow, 2) Data Flow and 3) Functional Dependency. Event flow consists of a source and a sink. An event flow is unidirectional. Events are sent and received by ports from both components. A data flow represents the flow of information. It is a continuous flow and is bidirectional. Any of the two components can initiate this kind of flow. A functional dependency is a dependency among the components where the components depend on a certain method or interface of the component.

The second group is called implicit dependencies. These include semantic constraints and integrity conditions, parameterization of components, changing the connections, adding and removing components. Implicit dependencies imposed no direct communication. These dependencies exist between services supplied by components and services stemming from basic underlying services such as memory, or scheduling, which may have an indirect influence on the performance of component.

Another work in this area was done by Keller and Kar [55]. They have presented an extensive model of dependency management. According to their model dependencies are classified in a multidimensional space of dependencies as shown in figure 2.3. The dimensions in this space are dependency criticality, dependency formalization, component activity, component type, dependency strength, space locality and domain. These dimensions are given in the figure 2.3.

The dependency criticality is the extent to which this dependency is important. It could be prerequisite, corequisite and exrequisite. Prerequisite are those services and components which are required before a component can be installed. Corequistes are components and services required in parallel with this component. Exrequisites are

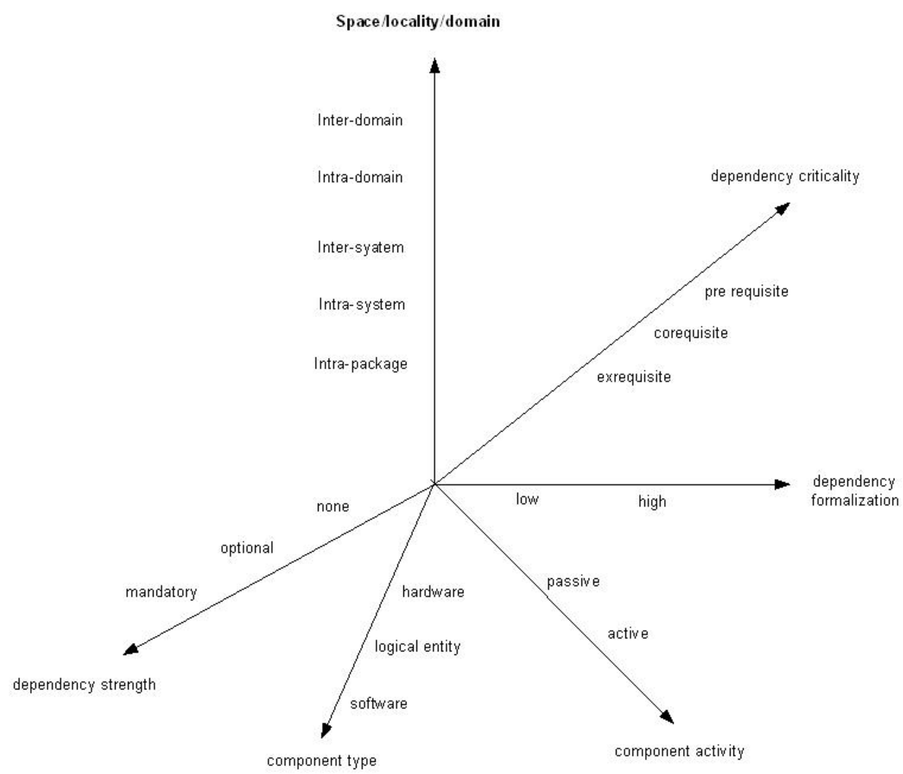


Figure 2.3: Dependency Classification by Keller and Kar

components and services which must not be present with a component.

The dependency formalization is the degree of formalization a dependency has. Because some information is available directly from the system properties. However, some dependencies may just be in the system administrator notebook.

The dependencies strength is how strongly the component is dependent on its antecedent resource.

The component activity records if a component is active or passive. An active component can be queried. However, a passive component cannot be queried.

A component type can be a hardware or software resource, a service or a logical entity.

Space locality and domain specify how far the antecedent is from the dependent. This is also referred to as interdomain vs intradomain and intersystem vs intrasystem.

2.1.4 Fault Tolerance

Software fault tolerance is the use of techniques to enable continued delivery of services at an acceptable level of performance and safety after a design fault becomes active [106].Fault tolerance is an area which improves the reliability of computer systems. The goal of fault tolerance is to reduce mean time to fail (MTTF) so that dependability improves in computer systems.

In 1986 Jim Gray presented his paper on Tandem Computer System. In that paper he identified software ad administration to be the major factors in the failure of computer systems. To mitigate these failures, he suggested a few mechanisms to incorporate fault tolerance in computer systems. To improve hardware fault tolerance he suggested a combination of modularity and redundancy. Moreover, he suggested that hardware must show a fail-fast behavior i.e. either it does the right thing or stops [48].

To mitigate administration related failures he suggested systems with minimal maintenance and minimal operator interaction. He suggested that systems must provide

a consistent and easy-to-use interface to the administrators.

To improve software maintenance, he suggested modular software design such that a failure in one module is not propagated to other modules in the system. Also, he suggested fail-fast behavior in software modules. Moreover, he suggested methods like checkpointing of different kinds to recover software after a failure. Moreover, he suggested ways to improve fault tolerant communications and fault tolerant storage.

Broadly, software fault tolerance is divided into two major categories: single version fault tolerance techniques and multi-version fault tolerance techniques.

Single version fault tolerance is based on use of redundant single version of a piece of software to find and remove software faults. Some of the relevant fault tolerance techniques include error detection, checkpoint and restart and data diversity.

Effective fault tolerance require two properties: self-protection and self-checking [1]. Self-protection means that the component must be able to detect any abnormal behavior of other interacting components while self-checking means that system must be able to detect internal errors and do appropriate actions to correct them.

Error detection checks vary with the requirement of software. Some error checks include replication checks, timing checks, reversal checks, runtime checks and so on. It is impossible to detect each and every fault in the system using these checks. Therefore, fault trees have been proposed. Fault trees can be used to identify general classes of failures and conditions that lead to those failures [15].

Apart from error detection, checkpoint and restart is another technique to incorporate fault tolerance in single version software. This is mostly a recovery technique. The most simple recovery technique is restarting a module or component. Static and dynamic restarts are possible. In static restart the goal is to return the module to a pre determined state while in dynamic restart the goal is to return the module to a dynamically inserted checkpoint. However, in some cases this kind of recovery is not possible. An example of this is messages that are sent to other modules. If a message

is sent there is no way it can be brought back [106].

In multi-version fault tolerance, two or more versions of a piece of software execute in sequence or parallel. These versions are used as alternatives to each other. The reason behind such a scheme is that if one software fails then the other continue to work because they both are built differently [11]. Some of techniques of multi-version fault include recovery blocks and n-version programming.

Recovery blocks technique uses the checkpoint and restart technique [83]. If an error is detected a different version of the software is tried. Checkpoints provide recovery to a previous state if one version detects an error. A primary version is used to provide complete performance while another backup version provide degraded performance of an error is detected.

Similarly n-version programming is also used to provide fault tolerance in muti-version software. Different version of software gives output to a given input. A decision algorithm is used to select the best output after evaluating all outputs. Considerable development efforts are required to build n-version programming. Moreover, the algorithm for the selecting the best output also require considerable effort [11].

Another variation of recovery block and n-version programming is N-self-checking programming [64]. In this technique both recovery block and n-version programming is used. However, an acceptance test is added to test the output of each block. Other variations in this technique have also been suggested through consensus recovery blocks [88] and t(n-1) version programming [107].

Cristian presented a classification of failure and related aspects from a distributed systems point of view. According to him a server failure occurs when it no longer behave in an expected manner. He divided failures into different kinds: timing failure occurs when the servers response is correct but does not reach at the specified time, response failure when the server does not responds correctly. Crisitan also presented taxonomy of crashes possible in the server. An amnesia crash is when the server starts in a pre-

defined initial state. In this crash the state does not depend on the state of the server before the crash. A partial-amnesia crash when some state is maintained after the failure but some state resets itself. A pause-crash is when the server starts in a state that it had before the crash. A halting-crash occurs when the server never restarts [32].

Cristian also presented semantics of failures in servers. Moreover, he presented an overview of the issues in designing hardware and software tolerance. In particular he discussed the issues related to modularity and redundancy like server groups.

2.2 Related Work

There are a number of approaches to failure recovery in software systems. Figure 2.4 presents a classification of the failure recovery techniques that we studied. Most of these approaches can be divided into three catagories: control theory-based techniques, architecture-based recovery and recovery oriented computing. Some of the techniques also overlap more than one catagory. Furthermoew, some approaches cannot be catagorized under these three catagories. Therefore, we present them under 'miscellaneous' catagory.

2.2.1 Control Theory-Based Techniques

Kephart and Chess presented an idea of autonomic computing [56]. Autonomic Computing is a computing paradigm where systems manage themselves given high-level objectives from administrators. There are four properties in these systems: self-configuration, self-optimization, self-healing and self-protection.

In self-configuring the systems configure themselves automatically through high-level policies that represent business objectives. When a new component is introduced in the system, it is added seamlessly with the rest of the system and the rest of the system adapts to its presence.

In self-optimization the systems continually seek ways to improve their opera-

Classification of Failure Recovery Techniques

Control Theory	Architecture Based Recovery	Recovery Oriented Computing	Miscellaneous
Kephart03	Oreizy99	Brown01	Klemm01
Knight01	Dashofy02	Sultan05	Koopman03
Reilly02	Tichy05	Serrano05	Oppenheimer03
Diao05	Georgas05		Garlan04
Park04		Patterson02	George02
Shin05		Candea04	
Garlan02		Candea03	
Arshad04	Cheng02	Arshad04	

Figure 2.4: Related Work Classification

tion. The improvement may be a performance increase or some operation that reduces resource etc.

In self-healing the systems detect, diagnose and repair localized problems that result in bugs or failure of the systems. Systems match the problem with diagnosis and apply the diagnosis followed by a test of the system.

In self-protection the systems protect themselves against the malicious attacks. Moreover, they also adapt themselves with credible early reports of problems.

All of these four properties of autonomic systems are based on a feedback loop of monitor, analyze, plan and execute. An autonomic manager controls these four phases.

The work in this dissertation has a number of properties of autonomic system. Especially our system performs self-configuring and self-healing. Our system performs automated failure recovery. Therefore, it has the property of self-healing. Moreover, our algorithm of failure recovery uses other available resources in the system to recover failed applications. Since, other resources may not be configured properly to host a given application, they need to be reconfigured. This reconfiguration is performed automatically in the system and the rest of the system adapts to the change in the reconfiguration. Therefore, our system also exhibits the self-configuring property.

Another important characteristic of the autonomic system is that configuration goals can be defined at a higher level and the system performs the rest of the low level operations to bring about the desired change. In our system the goals are very simple and defined at a very high level. It is the planner that searches the low level actions that bring about the desired change in the system. Therefore, our system has the property where the goals are specified at a high level and the system performs the low level actions. Many approaches to failure recovery have been suggested based on this feedback loop approach.

Knight and colleagues presented a survivability framework for software systems named Willow [60]. Willow is based on the concept that reconfiguration is required

at both the network level and application level for a system to achieve survivability. Willow supports both proactive and reactive reconfigurations. Willow uses Siena publish-subscribe mechanism for communication between the artifacts of the system. The reconfiguration is performed as a workflow and it is applied using the software dock framework.

Gurguis et al. presented an Architectural Runtime Configuration Management (ARCM) for failure recovery [45]. ARCM provides three capabilities. First, ARCM provides a monitoring and recording facility to record any changes in the architecture at runtime. Second, it provides a visual representation of the architecture. In order to achieve the above mentioned capabilities the architecture is represented as a directed graph where nodes represent architectural configurations and edges represent the possible path to these configurations. Third, it provides a recovery facility in the architecture. This recovery feature is provided through a rollback and rollforward operation in the architecture. Rollback is the process where the architecture is reverted to a previous configuration and rollforward is the process where the architecture is transitioned to a subsequent configuration in the version graph.

Diao et al. also used control theory for self-healing and self-managing systems [36]. They compare the similarity of control theory and self-managing systems. The authors note that the autonomic systems focuses on specification an construction of components that interoperate well for management tasks while the control theory focuses on algorithms and developing components that achieves the control objectives. Overall the authors specify four attributes of control systems that can be applied in self-managing systems to make it more robust. These attributes are stable, accurate, have short settling times and does not overshoot (SASO). Then they give the details of how the resource dynamics can be modeled as control theory problem and how sensors and effectors are developed that can use the control theory concepts in autonomic computing systems. Finally, in the paper the authors say that it is difficult to test the applicability

of control theory without an end-to-end system. Therefore, they developed a Deployable Testbed for Autonomic Computing (DTAC). DTAC can emulate a webservice and can be used to test the autonomic systems and the application of control theory concepts applied to it.

Reilly et al. also presented a control-based approach to application management and adaptation [84]. The authors describe a three layer model to achieve this. The top layer is the actual application layer where all the application middleware such as Jini are hosted. The second layer is the heart of their approach. It performs three functions 1) Instrumentation services for monitoring and logging the access and invocations. 2) Control services that examine the information obtained by the instrumentation services to identify a conflict and then select an appropriate adaptation strategy. 3) Dependency management layer that takes care of the up to date representation of the dynamic dependencies in the system.

This paper does not discuss how self-healing or failure recovery can be performed using the framework described in the paper. However, the idea of the control loop is what we are using in our dissertation for failure recovery.

Garlan and Schmerl presented an architecture-based approach of self-healing systems [47]. They argue that in current systems the adaptations are tightly integrated with the code of the system and cannot be changed easily. Therefore, they propose an externalize adaptation approach. In this approach they view the system under a closed loop control system which has the following phases: monitoring, abstraction, reification and adaptation. The system is continually monitored against some given models. When system fall outside these models then the system adapts itself to bring it back into the given boundaries of the evaluation models.

Shin and Cooke presented a self-healing mechanism using connectors [90]. The system model that the authors are using is of components. However, these components are made up of objects. Objects communicate with each other through connectors. This

paper focuses on the self-healing inside components, based on connectors between the objects. Each components is made up of two layers: A service layer and the healing layer. The healing layer reconfigures the objects in the service layer if its finds an object that is sick. Moreover, the healing layer also notifies the neighboring components about the reconfiguration it is performing. This step is to ensure that the neighboring components also reconfigure themselves if required. The mechanism of self-healing works as follows. The connectors notify the healing layer about the arrival of messages from objects. If there is an anomalous behavior in the arrival of these messages the component reconfigures objects and repairs the object. This healing is performed through the healing layer of the component.

Park and Chandramohan described two models of failure recovery: Static and dynamic [78]. The main strategy behind the static model is redundancy. A monitor maintains a list of available servers and their operating status. As soon as a failure occurs in one server the monitor switches to another server in a isolated setting. Moreover, in order to support the transactions that are currently going on two modes start an continue are specified.

In the dynamic recovery model, failed components are replaced by dynamically generated module which are deployed on the fly. The monitor can even generate immunized components to prevent further failures when the monitor knows the reason of failure. If the monitor does not know the exact reason of failure then it simply replaces the old component with the new one in a safe environment.

According to the authors the dynamic model is more efficient as it only generates the component when required. Therefore, it has lower maintenance cost than the static model.

2.2.2 Architecture-Based Recovery

Various approaches to architectural-based recovery are presented.

According to our literature survery, Oreizy et al. presented seminal work in the area of self-adaptive systems [75]. According to this paper a "Self-adaptive software modifies its own behavior in response to changes in its operating environment". It provides a spectrum of self-adaptability which ranges from conditional expressions to artificial learning.

This paper provides two different but related aspects of self-adaptation: Adaptation Management and Evolution Management. Adaptation management refers to the process where a system observes its own behavior and analyzes the observations to determine appropriate adaptations. There are four distinct steps in this process. 1) Collect observations is the process where the system is being monitored for some specified properties which include but are not limited to performance monitoring, safety inspections and constraint verifications. 2) Evaluate observations is the process where these recorded observations are analyzed and compared against an expected behavioral model. 3) Planning change is the process where changes are planned. 4) Deploying change descriptions is the process where the changes in he system are being propagated to the different sites so that agents can bring about the desired change.

The other aspect of self-adaptation that the paper discusses is evolution management. Evolution management focuses on changes in the structure of the system. It is different from adaptation management in the sense that in adaptation management changes are only applied on the existing system without changing the structure of the system. In evolution management the system undergoes a change in the structure by adding or removing existing components. The structure of the system exists in the form of an architectural model. This architectural model consists of connectors and coarse grained components. Components are responsible for implementing application behavior and maintaining state information while connectors are transport and routing services for messages and objects. The evolution is carried through an architectural evolution manager which maintains consistency and system integrity. The paper presents C2 and

Weaves as two dynamic architecture systems which perform evolution management.

Overall this paper provides some nice settings for this dissertation. Our lifecycle of Sense- Analyze -Plan-Execute is a way similar to the adaptation management. However, in failure recovery the structure of the system might also change. Therefore, evolution management is also performed in this dissertation. But this evolution management is in terms of adding removing components whose behavior is known. No new types of components are being inserted in the system.

Dashofy et al. presented an architecture-based solution to self-healing systems [34]. The authors describe four capabilities in the architecture if it needs to be reconfigured at runtime. First, the capability to describe the current architecture. Second, the ability to express an arbitrary change to the architecture, Third, the ability to analyze the result of the repair and four, the ability to execute the repair plan. For describing the architecture the authors have used xADL 2.0 which is an XML-based architecture description language. For expressing the architectural change a tool called ArchDiff is used. This tool has the capability to take two xADL 2.0 architecture descriptions and outputs a difference in the architecture based on the addition or removal of components. Moreover, an ArchMerge tool is used to merge two architectures and apply the difference of the architecture from ArchDiff to add the add-on artifacts to the base architecture.

In order to validate the repair results various kind of checks are performed on the new architecture. Most of these checks are performed by a what-if analysis. This analysis is performed off line and if the change is satisfactory then a planning agent will perform the changes to the actual system.

An Architectural Evolution Manager (AEM) is used to perform the changes to the runtime system. It allows the components and connectors to be replaced to execute a clean up code. Components and connectors bordering the affected area are suspended. Components, connectors and links are added as defined in the new architecture. Following that the components and connectors bordering are resumed and the communication

is started.

Tichy and colleagues discussed a rule-based technique for failure recovery in component-based systems [97]. They developed this technique such that components themselves find self-repair actions that can minimize damage and reduce repair time. The objective function of a failure is presented as an inequality. This inequality is solved using an algorithm which includes a presolving phase and a submodel expansion phase. In the presolving phase the properties of the system not really relevant to the failure are removed to speed up the process. The submodel expansion uses an algorithm to calculate the actions required to recover the system from failure.

Cheng et al. used architectural styles as a basis for system self-repair [28]. In this technique they made the architectural style a first class entity. Using this, they determined about the properties of the system to be monitored, constraints that need to be evaluated and so on. Moreover, the style can be used to select appropriate repair strategies.

2.2.3 Recovery-Oreinted Computing

Recovery-orinted computing is yet another area of failure recovery. The goal of recovery orinted computing is to reduce the mean time to repair instead (MTTR) of reducing mean time to fail (MTTF).

Brown and Patterson argued that fault tolerant techniques are not enough to ensure high availability of systems [17]. They say that hardware, software and the people who operate them all make mistakes. Therefore, failure recovery and repair must be considered a vital aspect in systems. They argue that the systems present today are heterogeneous and much more complex than homogeneous systems like transaction processing systems. Therefore, availability in these systems cannot be achieved by simply making the system more fault-tolerant but there is a need for a new paradigm required. They call it Recovery-Oriented Computing (ROC). In ROC, failures in the

system are accepted as a normal unavoidable fact of system operation. Therefore, the systems are designed to provide fast and effective recovery and repair mechanisms.

Patterson and colleagues argue that there is a lot of emphasis on performance improvement in recent years [79]. Therefore, computers are many folds faster now. However, this performance improvement has neglected other critical aspects which include total cost of ownership, dependability, privacy, etc. Moreover, emphasis on performance neglected another important aspect in computers and that is availability. According to Patterson and colleagues operator errors account for a large percentage of failures in computer systems. In order to get ideas for improving availability of computer systems, Patterson and colleagues looked at other fields to find ideas and inspirations to improve availability. In this paper they discussed three such field: disaster and latent errors in emergency systems, human error and automation irony and civil engineering and margins of safety. They concluded that to improve availability, we need to repair fast instead of recover fast. Therefore, their focus is to improve mean time to repair (MTTR) instead of mean time to fail (MTTF).

Candea and colleagues presented a paradigm of failure recovery called Recovery Oriented Computing(ROC) [20]. The paradigm of ROC is different than fault tolerance as ROC does not try to prevent faults but instead finds ways to recover the system. On the other hand fault tolerance try to prevent the fault as much as possible.

To incorporate recovery from failures ROC has two types of methods. Microreboot and Undo/redo. When a system fails there is usually a part of the software that fails. Sometimes these failures can be corrected by simply rebooting the whole machine or the software. The authors presented an even minuscule scale of rebooting. This rebooting is performed on the part of the software that behaves incorrectly. According to the authors the microreboot can be performed in a fraction of a time than the full rebooting.

The second method for ROC computing is undo/redo. In this method the opera-

tor are given the opportunity to undo or redo an action if the effects of the action cause a problem in the system leading to a failure.

The techniques presented in this paper are for a particular kind of failures which are caused by transient hardware and software failures. Moreover, they can also be effective against resource leaks and volatile data structures.

Candea and Fox presented a different paradigm for failure recovery [21]. They call it crash-only software. In this paradigm the software does not go through the phases of graceful shutdown. Instead its stopping is equal to that of a crash. This is because the crash reboot of a system takes less time than clean reboot. They argued that a lot of effort has been wasted to make the software shutdown gracefully. However, these efforts are not really required if some changes are made in the software. One of the changes they suggested to make the software crash-only is not to store the state of the software in the middle tier. The stopping must be induced from outside. The components in the system must have a clearly defined boundaries. Moreover, all interactions between the components have a timeout. Furthermore, the requests are self-describing so that they do not depend on any session data. Incorporating these properties will make the software crash-only. They say that doing this will make the software easy to model and recovery time will take less.

Human operators often perform an action on system that they need to undo. Serrano et al. presents a method for correcting these errors through indo operation in Management Information Systems (MIS) [89]. This undo feature in MIS systems stores the tables in the database. Any historic data could be produced using these historic tables. Therefore, if any user of the system performs an operation to the data that he or she wishes to undo, he or she can recover the data from the historic storage of the data.

The crash of an operating system can cause a loss of the system state. However, in stateful services this session data is used by the user of the system. Loss of such data can

cause inconvenience to the user of the system. Sultan et al. provided a mechanism to recover such sessions using a backdoor architecture. In ordinary cases such session data is only accessible from the memory through operating system of the same machine. Backdoor architecture provides an alternate route to such data. Sultan et al. used a Ethernet network interface card to get access of the memory of the system where operating system crash occurs [95]. This access is performed by the machines in the same cluster running the same applications. Therefore, as the applications are the same the data from the failed machine is retrieved and the client does not see a much delay in getting his/her session back. According to the authors a restart could also recover the operating system but a restart is destructive to the transactions already taking place. Moreover, a restart involves a significant downtime compared to backdooe architecture. The authors modified an operating system to include backdoor architecture. The first modification is the ability to detect a failed operating system and the second, is to include the capability of service continuation when one of the peer operating system fails.

2.2.4 Miscellaneous Approaches to Failure Recovery

Some other approaches are also present in the literature that cannot be out into the other three catagories.

George and colleagues presented a self-healing approach based on the biological phenomena that exists in organisms [46]. They describe four properties that make a system self-healing. Environmental awareness, where the cells sense the properties of the surrounding environment e.g. like any damage or any other abnormal behavior. Adaptation, where the cells even with the failure of some cells develops an organism or a part of it. Redundancy, where many cells are devoted to the development of a single organism and the death of a few cells does not stop the development of the organism. Decentralization, where the cells can induce their neighboring cell to perform a certain

task.

The authors have described three approaches that have self-healing properties and then used these properties to design their own programming model for self-healing systems. The three ways of self-healing that are found in organisms are Morphogenesis, where cells adapt to the environment if one of their neighbors fails. Wound healing, where the layers underneath the wound responds to the failing of the upper layer i.e. the skin and performs inflammatory actions to protect the cells below it. And finally Regeneration, which uses either stem cell, based approach to regenerate or uses a cell multiplication approach.

The authors have described a simulator where they have simulated these behaviors to design their own simulations. The three biological phenomenons that they mimic are cell actions, cell division and gene actions.

Garlan et al. argues that there are situations in the self-adaptive system where priority needs to be given to one or more factors [43]. As the user is the person who has to benefit from the system, there must be a task aware adaptation of the system. In this kind of adaptation the system store an explicit representation of the user intent and base its adaptations based on the desires of the user. In the architecture described in the paper, there are two layers above the environment of the system. A task management layer stores all user preferences and detects any changes in those preferences. A environment management layer translate the user intents into the changes in the environment through configuring the elements that make up the environment.

Klemm and Singh presented a Java server specific technique for failure detection and recovery [59]. They claim that Java language provides many fault tolerance mechanisms like exception handling, garbage collection and so on. Moreover, Java unlike C/C++ provides a graceful degradation mechanism through its exception handling mechanism but still the availability of these servers are affected. Therefore, they proposed a Java Application Supervisor (JAS) that is a an external application to Java

Runtime Environment (JRE). JAS provides automatic failure detection and recovery. A user specifies policies about what to do if a failure occurs. Based on these user policies, divided into thread and system policies, JAS take the appropriate action to recover from failures without human intervention. JAS is not part of JRE. Thus, the failures are handled externally without modifying the JRE.

Oppenheimer et al. presented a study of three internet services and examine the failures that occur in them during the course of five months [74]. They are highly accessed internet services and two of them have a hit rate of approximately 100 million.

The authors noticed that the failure due to operator errors surpass all other kinds of failures. Most of the operator errors are due to misconfiguration and similar mistakes. The authors also examines failures that occur in components and results into service failures later.

The authors also provide a way to mitigate these a failures through various techniques which include correctness testing of components, redundancy, configuration checking, etc.

Chapter 3

Motivating Example

The goal of this dissertation is to automate the failure recovery process of the systems that provide internet-scale services [74], such as those provided by Google[1] and Yahoo[2] . Due to the 24 X 7 operation of these systems, they have very high availability requirements. Even a small downtime may cause huge revenue and opportunity loss [37]. For example a single hour of downtime can cost up to $200,000 to companies like Amazon and up to $6,000,000 to brokerage firms [37].

Internet services like Yahoo and Google have a huge infrastructure and are hosted on a myriad number of machines. Moreover, due to the proprietry nature of these services it may not be possible to evaluate the techniques in this dissertation to the scale of the systems providing these services. Therefore, in this dissertation we have selected off-the-shelf components to mimic the behavior of internet services at a small scale.

3.1 Reasons of Failures

There are numerous reasons for the failure in these large-scale distributed systems. Brown and Patterson have identified three significant reasons for this downtime: hardware, software and human operators [18].

[1] http://www.google.com
[2] http://www.yahoo.com

Most of the internet-scale services are hosted on cheap failure-prone commodity hardware instead of the expensive failure-resistant hardware [18]. Due to the lack of fault tolerant components in commodity hardware, like the unavailability of ECC memory, less built-in fault tolerance is available. Therefore, machine failures cause a huge chunk of the downtime. Patterson and colleagues measured that hardware errors cause 15% of the downtime in these systems [79].

Apart from hardware, the component and service failures also are a reason for this downtime. Component failure are usually manifested by faults. These faults usually come from custom written software deployed on these components [74]. As components provide services to the outside world the failure of a component results in the failure of service also. According to Patterson and colleagues, software failures account for about 34% of the total downtime [18].

However, the biggest cause of the downtime in these services is not hardware or software failures. In fact it is the operator errors that cause 51% of the downtime. These systems are huge in size and complexity. Therefore, administrators of these systems have a difficult time maintaining them. Most of the time it is not possible to undo change right away; therefore, the time to recover the system is significant. A large portion of operator errors are due to misconfigurations in the system [79].

3.2 A Typical Internet-Service System

A typical internet service consists of three layers: A front end layer, usually called a presentation layer, a middle tier that contains the business logic and then a database layer for persistent storage. Moreover, there are systems with two layers where the front end layer and the middle tier are combined. Furthermore, systems with four and more layers are also common. In these systems there is an additional load balancer on top of the front-end layer for load distribution [74, 24].

The front-end in a typical three-tier system is usually a webserver like Apache

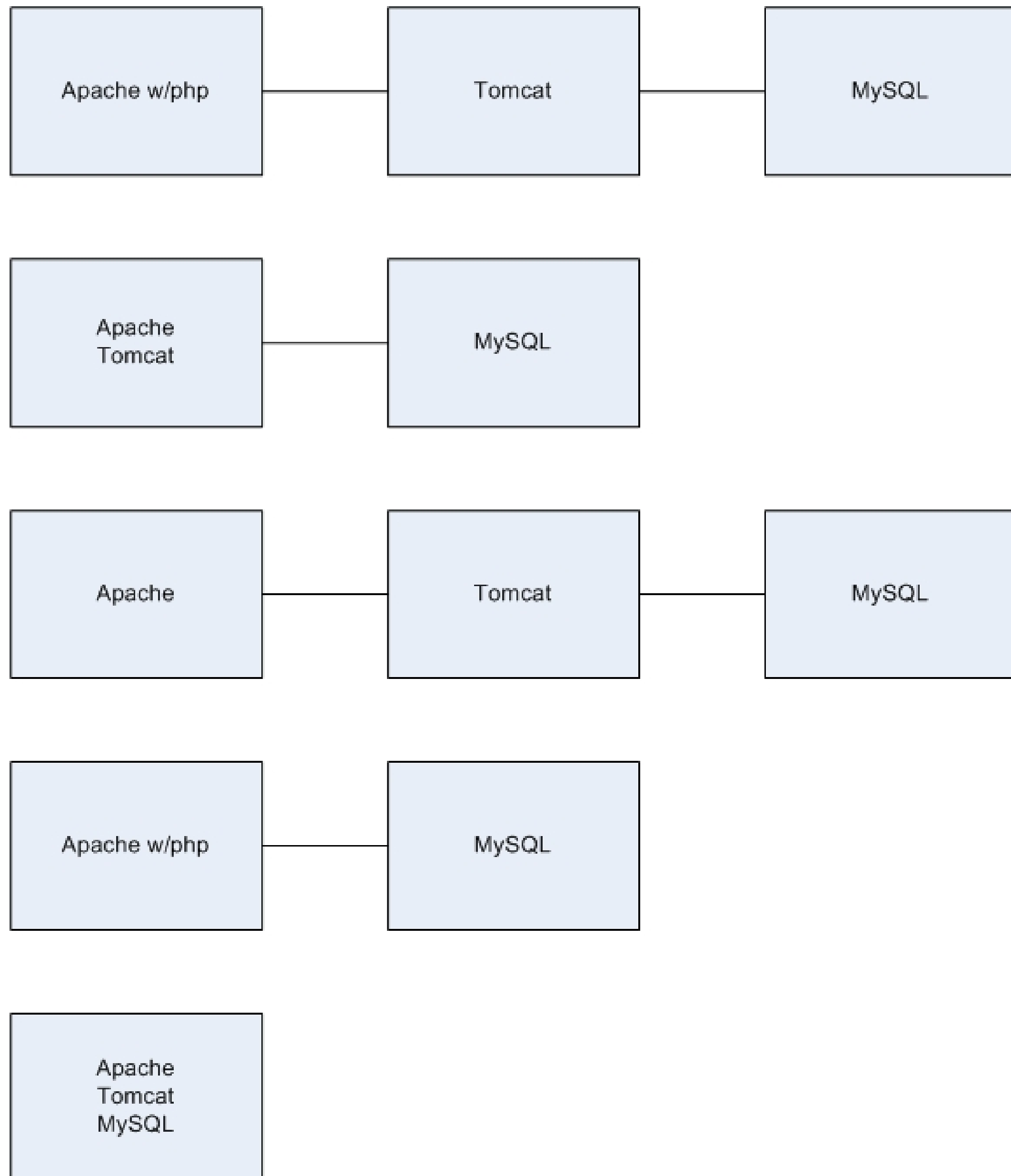


Figure 3.1: Various Configurations of Internet-Scale Systems

httpd server or Microsoft SQLServer. The middle tier consists of either a servlet engine such as Tomcat or an application server such as IBM Websphere or BEA Weblogic. A combination of servlet engine and application server is also present in some architectures. The database tier consists of a database such as Oracle or Mysql.

The front-end receives requests from clients. The front end usually processes this request and pass it on to the middle tier. The middle tier contains the business logic of the system. The middle tier processes the request and access data from the database tier, if required. It then returns the processed data to the front-end which presents the data back to the client. The presentation of the data in a meaningful manner is also a job of the front-end.

In this project we are using distributed systems with three layers with Apache httpd server at the front-end, Tomcat servlet engine as the middle tier and Mysql database as the database tier. Using Apache, Tomcat and Mysql for our project seems to be the logical choice since they are openly available with their respective source code [36]. Apache is also a very popular webserver. According to a recent survey by Netcraft, more than two third of internet websites are using Apache [72]. Tomcat is also a very popular servlet engine it can work in conjunction with Apache or can recieve client requests directly. However, a configuration of Tomcat with Apache gives a better performance than a configuration with stand-alone Apache. Mysql also is very poluar database. It is available for free and it stores all the data in the system. From this point onwards we will use names of Apache, Tomcat and Mysql for the front-end, middle tier and database tier respectively.

Internet-scale systems can come in a variety of architectures in terms of placement of components. Some example architectures of these systems are given in Figure 3.1. The boxes in this figure represent machines. The three components Apache, Tomcat and Mysql can be placed in a number of ways on these machines. Each sub-figure in figure 3.1 represents a way of placing these components. These components can be places

with each other on a common machine or they can be placed on seperate machines.

Apache, Tomcat and Mysql all come with a default configuration. They provide some basic services with their respective default configurations. In case of Apache the default service is http service and in case of Tomcat it is the servlet service. However, both of these servers can provide additional services such as php service or cgi service if additional modules are added and configured. Configurations in all of these servers are stored in one or more configuration files. These configuration files are modified or in some case more configuration files are added to install additional modules. To communicate with each other, these servers also have modules called connectors such as jk or jk2.

Applications are deployed on these servers. A particular configuration of an application may require any combination of these servers. If an application requires default services then the default configuration of these servers is enough. But in cases where applications require extra services then these servers have to be reconfigured before an application could provide useful service to the user.

An application in an internet-scale system is usually a website or portal. Websites are constructed in a myriad number of ways. Different technologies and tools are used to develop these websites. For example, a website may use Cascading Style Sheets (CSS) at the presentation layer and Java Server Pages (JSP) for the business logic layer. Another website may use simple html for the frontend and servlets at the business logic layer. Yet another website may use CGI scripts or PHP at the front end and combine the presentation layer and business logic layer. Some websites are solely deployed in the business logic layer i.e. Tomcat and they just use the Apache as a proxy for forwarding requests to Tomcat. Hence, one can see the large variety of tools and technologies used to develop a website application. It is important to note here that the technologies like CSS, PHP and so one require specialized services from the components like Apache and Tomcat. These services are not provided by default and needs to be installed and

configured on these components.

Mysql also requires a configuration to work but in default cases the configuration of Mysql is usually simple and does not require a lot of manipulation in the configuration files. Moreover, databases systems such as Mysql already have built in failure recovery mechanisms. Therefore, in this project we assume that the configuration of the Mysql database is fixed and it does not require manipulation in cases of failure recovery.

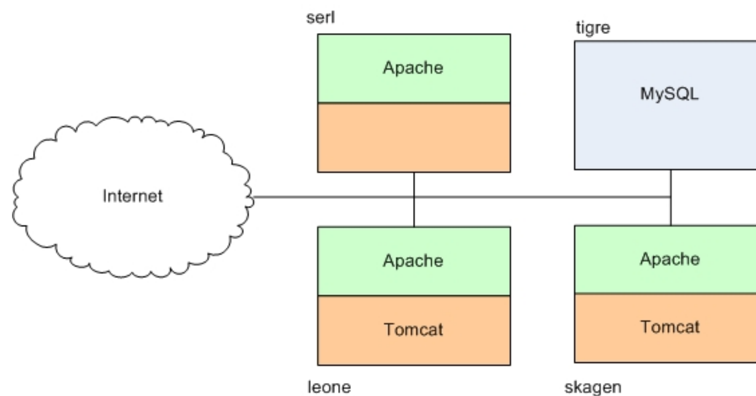


Figure 3.2: Architecture of System used in this Dissertation

3.3 Distributed Systems Architecture

In this dissertation we are incorporating automated failure recovery mechanism in an architecture which consists of three websites or applications. These three applications are Sms, Rubbos and Webcal. Sms[3] stands for Strand Map Service. It is a service designed to provide strand map functionality to the user. It is an educational resource for Digital Library for Earth System Education[4] . This particular service provides a graphical interface for the strand maps [96].

Rubbos[5] is a bulletin board benchmark site designed like slashdot with huge amounts of data. People can post messages on this bulletin board and all these messages

[3] http://www.dlese.org/Metadata/strandmaps/
[4] http://www.dlese.org/dds/index.jsp
[5] http://www.cs.rice.edu/CS/Systems/DynaServer/RUBBoS/

can be accessed by other people.

Webcalendar[6] (called Webcal also) is an online calendar for individuals and/or groups for scheduling meetings, tasks etc.

All these applications have a set of requirements in terms of dependencies. Morever, the authors of all these applications provide more than one high-level configuration for their deployment. These deployment configurations are specified at a very high-level and can be used as guide to deploy these applications. However, these applications have to be configured at each component where they are deployed seperately.

The various configurations of these applicatios are given below.

Sms (Strand Map Service)

Possible Configurations

(1) Apache + Tomcat + Mysql

(2) Tomcat + Mysql

(3) Apache

Rubbos

Possible Configurations

(1) Apache + Tomcat + Mysql

(2) Apache/PHP + Mysql

(3) Tomcat + Mysql

(4) Apache

Webcal

Possible Confgurations

[6] http://sourceforge.net/project/showfiles.php?group_id=3870

(1) Apache/PHP + Mysql

(2) Apache/PHP

Not all the configurations of these applications provide the full functionality. The general rule is that the lower the number, the higher the functionality provided by the application. Therefore, all the applications are deployed in the system with configuration 1. If a failure occurs the goal is to restore configuration 1. If this is not possible a lower number configuration is applied.

In order to test and evaluate our failure recovery mechanism described in this dissertation we have used four machines: serl, skagen, leone and tigre. Three Apache components, two Tomcat components and one Mysql component are present on these machines. The placement of all these components is fixed during the lifetime of the system. The architecture of machines and their various components is given in Figure 3.2

Chapter 4

Modeling for Failure Recovery

In this chapter we will discuss the modeling issues of the systems described in the previous chapter. Modeling such systems gives us the ability to incorporate automated failure recovery techniques in them. However, due to the various architectures available for developing distributed systems it is very difficult to develop a failure recovery model that is applicable to all types of distributed systems [62]. Therefore, in this dissertation the modeling is geared specifically towards the internet-based distributed systems using components like Apache, Tomcat and Mysql. Although we are modeling these components, the essence of this modeling can be used in any other similar type of distributed system.

Every modeling starts out with a theme or requirement. The requirement of modeling in this dissertation is **functionality for the user**. It is our goal that whatever is going on behind the scenes the user must get functionality back as soon as possible. Therefore, in any case failure recovery of a part of the system that is never used must not precede a part of the system that is used frequently by the users.

Functionality for the user is provided by an application. The application is deployed on a component. However, it does not matter what component the application is deployed on as far the user is concerned. The concern for the user is the functionality of the application with which he or she is interacting.

As applications provide functionality to the user, the driving force of our failure

recovery modeling is an application. We define an application as *"a software entity that is capable of providing direct functionality to the end users of a system"*. A common example of an application in our architecture is a website. A website provides a direct functionality to the user while hiding any details of where and how it is deployed. An application cannot work alone. It is deployed on one or more components to provide the intended functionality. Each application requires the component to provide some services. Without these services the application cannot provide the functionality to the user.

As components provide the services to the application, components are the next artifact in our modeling process. We define a component to be *"a software entity that hosts an application and provides the necessary services that are required for the working of the application"*. Apache, Tomcat and Mysql are all considered components in our model. In fact, these three are types of components and more than one instance of them could be running at a given time. A component requires a machine to be able to work. A component has a default configuration but it can have additional configurations also. But the additional configurations are driven in part by the needs of the application being deployed. The configuration of the component is changed through manipulating the configuration files.

Components provide services. Some services are provided by default and additional services are provided through additional modules. Components connect with other components. In order to connect two types of components, a special type of module called a **connector** is required. The connector provides a communication mechanism between the components.

Components are deployed on machines. Therefore, machines are also part of modeling for failure recovery. We define a machine as *"a hardware entity that has certain resources like memory and hard disk and contains a operating system to work with other software entities"*. Components are deployed on machines. Depending on

the type of component, more than one instance of a component can be present on a machine.

These three artifacts - application, component and machine - have numerous properties. To automate the failure recovery process, we have formalized the properties of these artifacts into different models. Inside these models there are two kinds of sub-models. **Static models** remain the same during the lifetime of the system. The fixed properties of the artifacts are included in the static model. On the other hand **dynamic models** are continously updated during the lifetime of the system. These models capture the properties of the artifacts that change at runtime.

4.1 Application Model

To model the applications for failure recovery two separate models are needed: an application configuration model (ACM) and an application dynamic model (ADM). ACM is a static model and usually stays the same during the lifecycle of the application while ADM is a dynamic model and changes with any change in the actual deployment of the application.

4.1.1 Application Configuration Model

An application is deployed on components. An application can be deployed using multiple high-level topological configurations. For example, in the previous chapter we saw that multiple configurations of different applications are available and all of them had various dependencies and requirements. These configurations are fixed and cannot be changed. Therefore, these configurations are to be stored somewhere so that they can be used for failure recovery.

An application configuration model (ACM) is a repository of the possible configurations of each application in the system. Each configuration is identified by a configuration number. This model give details of dependencies of each configuration.

A template of ACM is given below.

Application Configuration Model

(1) Application Name

(2) Configuration Number

(3) Component A

  (a) Component ‘A Included
  (b) Component ‘A Intercomponent Dependencies
  (c) Component ‘A Intracomponent Dependencies
  (d) Application Import Time to Component ‘A

(4) Component ‘B

  (a) Component ‘B Included
  (b) Component ‘B Intercomponent Dependencies
  (c) Component ‘B Intracomponent Dependencies
  (d) Application Import Time to Component ‘B

(5) Component ‘C

  (a) Component ‘C Included
  (b) Component ‘C Intercomponent Dependencies
  (c) Component ‘C Intracomponent Dependencies
  (d) Application Import Time to Component ‘C

.. ..

Each configuration is identified by a configuration number. Moreover, an application requires certain components in each configuration. The information about the types of components required in each configuration is stated in this model. The detailed information about the component is given in terms of intercomponent and intracomponent dependencies of an aplication in each configuration. Note that this configuration information is specified at a very high-level. Low level details about a configuration are not included. Most of the applications may have only one such high-level configuration. The maximum configurations that we have seen in the applications that we are using in this dissertation is four.

The first sub-field in items 3-5 is an indicator of whether the component is part of the configuration or not. It is a boolean value. One (1) means that the component is part of the configuration and zero (0) means that the component is not part of it.

The rest of the three sub-fields give further information about the component only if the value of the first item is one (1). "Intercomponent Dependencies" give the connection requirement for the component. For example, in a particular configuration Apache needs to be connected with Tomcat. This fact that the application requires Tomcat to work is stated in the intercomponent dependencies item in the ACM. Note that the exact method of connection is not specified because it is a low level detail. It is up to the failure recovery mechanism to find compatible instances of components to connect with each other. The information about which type of connection a component supports is given in another model described later in this chapter.

"Intracomponent Dependencies" list the services that this particular configuration of the application requires from the component. If only default services are required then they are not listed. Only additional services are listed. An exmaple of an intracomponent dependency is the requirement of the application Webcal to support PHP in Apache. PHP is not a default service of Apache. Since it is not a default service it is listed under this item. Likewise, an application may require ssl service for secure connection. Since ssl is not a default service it is also listed here. Again this is a very high-level detail and the low level details which include the actual specification of these services are not listed here.

The last sub-field is the application import time. An application is imported to a component on a need basis. After a failure of an application it is imported in another component from a central repository. The time required for the application to be imported to a component is called "import time". Application import time is also fixed because the size of the application is fixed and cannot be changed during the lifetime of the system. One may argue that it can change if one machine is faster than

the other. However, in this dissertation we have used identical machines so this time is fixed.

More than one application could be present in one system at a given time. Therefore, to identify a configuration uniquely, which is the key to find a particular configuration of a particular application, both the application name and configuration number are required.

This model contains the information about configurations of applications in the system. A sample configuration for the application Rubbos as it appears in ACM is given below. This is the configuration 1 of Rubbos from the previous chapter. Other configurations are represented in a similar way in ACM.

(1) Application Name: Rubbos

(2) Configuration Number: 1

(3) Apache

  (a) Apache Included: 1
  (b) Apache Intercomponent Dependencies: Tomcat
  (c) Apache Intracomponent Dependencies: php-service, ssl-service
  (d) Application Import Time Apache: 12

(4) (a) Tomcat Included: 1
  (b) Tomcat Intercomponent Dependencies: Mysql
  (c) Tomcat Intracomponent Dependencies: jsp-service
  (d) Application Import Time Tomcat: 15

(5) Mysql

  (a) Mysql Included: 1
  (b) Mysql Intercomponent Dependencies: 0
  (c) Mysql Intracomponent Dependencies: 0
  (d) Application Import Time Mysql: 89

4.1.2 Application Dynamic Model

An ACM captures the static configuration information of applications. However, these applications are deployed on the system at runtime. The information about their state at runtime is captured in the application dynamic model (ADM).

At any given moment, an application can exist in only one configuration. This configuration may change during the lifetime of the application due to reconfigurations in the system.

Reconfigurations are initiated by an administrator or automatically after a failure. To support this dynamism in application configuration, ADM is used to track configuration changes in the application during the lifetime of the system.

Following is the template of ADM that captures the runtime information of an application.

Application Dynamic Model

(1) Application Name

(2) Current Configuration

(3) Component A

  (a) Component 'A' Machine
  (b) Component 'A' State

(4) Component B

  (a) Component 'B' Machine
  (b) Component 'B' State

(5) Component C

  (a) Component 'C' Machine
  (b) Component 'C' State

.. ..

ADM has the name of the application and the configuration number. The configuration number must be one of those found in the corresponding ACM of this application. Moreover, this model also contains information such as on which machine a component is working and what the state of the component is. Therefore, for each type of component a separate field is present in this model. It has the name of the machine where the component is present and a state indicator giving information about its current state. This state indicator could have three values: W, N or F. W means that the component

is working. N means that it is not working but an attempt could be made to turn it on while F indicates that the component is failed. If a zero is present instead of the name of the machine then it means that the particular component is not part of this particular configuration.

4.2 Component Model (CM)

A component model contains information about the components. It includes static information like the interfaces of each component and the dynamic information such as the behavior and state of the component at runtime.

4.2.1 Component Interfaces

Any component has a set of well-defined interfaces that provide management facilites to its users. Therefore, a list of these interfaces and their respective usage in management of components is required. Therefore, all the components that are used in the system must provide a list of these interfaces so that they can be used in failure recovery.

In our architecture Apache, Tomcat and Mysql are used. These artifacts were not developed originally as components. However, they have a lot of similarities with components. They provide management facilities to their users. Moreover, they have notions like state, type and so on. However, unlike components, they lack well-defined interfaces. Their management is mostly performed through a mix of manipulating configuration files and control commands. Thus, in order to model them as components, well-defined interfaces are required. Upon close examination, it is clear that they can be modeled as components with well-defined interfaces.

All of these components have the information related to their working in various configuration files. For example, in Apache, the httpd.conf file has most of the configuration information. Most of the configuration changes in Apache can be performed

through this file. Nevertheless, other configuration files are also present in addition to this particular configuration file. Similarly Tomcat has a server.xml file that has all the important configuration information related to the working of Tomcat. In configuration files the configuration information is specified through various directives. Modification of these directives indicates a configuration change for the component. Some configuration changes could be performed by just changing the value of a directive to another value. However, for some configurations some complex manipulation of the directives is required.

In order to change the configuration of Apache or Tomcat administrators modify these files manually. This manual process can take a long time if any failure occurs in the system. In addition, operator errors are the biggest cause of downtime in systems [18]. In cases of failure recovery, operators are sometimes under intense pressure to bring the system back online. Therefore, the chances of error are always there in failure recovery. More errors may do even more harm to the system that is still recovering from failure. All these factors add more time in failure recovery.

Our goal in this project to make the recovery process as automated as possible. Therefore, we have developed shell scripts to change the configuration of Apache. These shell scripts are atomic. Atomic means that they change the smallest level of configuration. The smallest level of configuration of the components in our system is a configuration directive. Each shell script modifies one particular configuration directive in these components. A script can take parameters, and based on the value of those parameters, change configuration file. With the help of these scripts an administrator can run the script to obtain a desired configuration. However, the running of the scripts is still a manual process. Furthermore, every failure scenario could be different and one or more scripts need to be applied to recover the system from a failure. The execution of these scripts cannot be random and they need to be executed in a certain order. Since a recovery script can be different for classes of failures, it poses a tough challenge for

the administrator to have a recovery script for each possible failure scenario. We have solved this problem by considering a script as an interface. Each script in our model is an interface to the component. Since these scripts have the property of an interface, i.e. a name and parameter(s), they can be treated as interfaces. Therefore, each script in our model is an interface with a specified number of required or optional parameters. We will discuss the mechanics of automated failure recovery using these scripts in the next chapter.

These scripts/interface are of two kinds: 1)Configuration Interface and 2)Control Interface.

Configuration Interface: The scripts that are used to change the configuration paramters of the component are called configuration interface. These scripts are used to make changes in the configuration files. Sometimes based on the need of the application new configuration files are also created using these scripts. Some examples of scripts used in this dissertation include "addNameBasedVirtualHost" script in Apache. This script is used when one instance of Apache fails. Since Apache has failed the application deployed on it is also considered failed. If another Apache instance is available somewhere in the system then it can be used to host this application in addition to an already hosted application. Therefore, "addNameBasedVirtualHost" is one of the interfaces provided by Apache to change its configuration. This type of interface is called a configuration interface.

Control Interface: All components in our project are standalone components. These components have to be started and stopped manually using control commands. These control commands form another category of interface in our modeling called control interface. This control interface is also part of the interface for each component. Start, restart and stop operations of a component are all part of the control interface. 'Start' means that a component is started from a stopped state. Stop means that a component is stopped form a started state. Restart means that the component is in a

started state but it is stopped and started again. Other states like hibernate, sleep and so on are also possible.

Configuration interface is different for Apache, Tomcat and Mysql since different kinds of configurations are possible in them. Therefore, each of them specifies a different set of interfaces to manipulate their configuration. On the other hand control interface is the same for all three types of components. In this dissertation we are only using the start, restart and stop interfaces provided by these target components.

4.2.2 Component Properties

Apart from interfaces component properties are also required to model the runtime behavior of components. These properties are updated regularly to ensure an up-to-date picture of the component. A template of the component model that captures the properties of the components at runtime is given below.

(1) Component Type
(2) Component Machine
(3) Component State
(4) Start Time
(5) Restart Time
(6) Accessible Port
(7) Module Currently Installed
(8) Module Available to be Installed
(9) Current Load
(10) Component Instance Number
(11) Connectors Available
(12) Applications Installed

Among these properties, "Component Type" determines the type of a component. The three types of components in our system are Apache, Tomcat or Mysql. "Component Machine" is the name of the machine where the component is deployed. The name

of the machine should be unique in the system. “Start Time” and “Restart Time” give the approximate amount of time a given component takes to start and restart. This time is stated in seconds. Usually this time is the same for components of the same type but could be different if the version and/or platform of the component is different. “Accessble Port” is the numeric value of the port where the component can be accessed through its respective protocol. For example, the default port for Apache component is port 80, for Tomcat it is port 8080 and for Mysql it is port 3306. Other ports are also possible in a given component.

“Component State” is an indicator attached to each component in this model. This state indicator could have three values: W, N or F. W means that the component is working. N means that it is not working but an attempt could be made to turn it on while F indicates that the component is failed. It means that the component cannot be restarted. This state can only be possible if the machine on which the component is deployed has failed. Therefore, the component deployed in this machine is also failed. Unless the machine is back online the component cannot be started. Another reason for state F is when repeated attempts have been made to restart the component but somehow due to a catastrophic error the component cannot be restarted.

In this dissertation a component is defined as a software entity that provides services to the application. Components provide these services through their modules. Some services are provided to the applications by default; others are provided through additional modules. For example, Apache provides the service of http access by default but to provide a php service, an appropriate module needs to be installed and configured as an add-on.

Three kinds of modules are present in the system: First, modules that are already installed in the component and they are providing their intended service; Second, modules that are not installed but can be installed if an application requires them; and, third, modules that are not available at all with the component. The modules of

the first and second catagory are described in the fields 7 and 8 in "Module Currently Available" and "Module Available to be Installed" fields respectively.

All the components are also under varying loads at different times. Therefore, to track the load on a component in a given time period ,a field "Current Load" is provided. The load is calculated as follows. Each component has a maximum number of concurrent connections possible at one time. Therefore, the average number of connection in a specified period of time divided by the total number of connections give the load of the component in a given time period. This load is specified in terms of percentage.

It is possible that more than one component of the same type are working on a machine at some time. Therefore, the component instance field provides more information about the identification of the component. The component instance is usually the process id if the components are deployed on unix or its variants.

Finally, each component has one or more applications deployed on it. In this dissertation the maximum number of applications that could be deployed on a component is two. The list of applications deployed on a component is in field 12 "Applications Installed" of the component model.

Each instance of components is identified by three parameters: type, instance and machine. These parameters are used to identify a component uniquely among the set of working components in a given system.

In brief, each type of component in our model provides interface for controlling its functionality. These interfaces are of two kinds: control and configuration. Control interface provide functionality such as start, stop and restart while configuration interface provides functionality such as add module, deploy application etc. These interfaces are fixed and are not changed during the lifetime of the system. These interfaces have parameters to modify the state or configuration of the component. Another important point is that the component is an atomic entity in our model. A component exists only in its entirety. It is not possible that a component exists on two machines. A component

can only exist at one machine at a given time.

4.3 Machine Model

Components are hosted on machines. Therefore, it is necessary to keep track of the state of the machines since a failure in a machine is undoubtedly a failure of all components installed on it. Therefore, the information in the machine model includes properties of machines such as the operating system, architecture, IP address, components installed and so on. Moreover, since machines always are under variable loads, the information such as memory available or hard disk space available and so on, is also part of this model. Furthermore, machine model also keeps track of the components installed on the machine, application installations available and the state of the machine.

The template of the machine model used in this dissertation is given below.

Machine Model

(1) Machine Name
(2) Machine IP Address
(3) Application Installations Available
(4) Maximum Load
(5) Current Load
(6) Total Memory
(7) Available Memory
(8) Total Harddisk Space
(9) Available Harddiak Space
(10) Operating System
(11) Machine Architecture
(12) Components Installed
(13) Software Available
(14) Machine State

“Machine Name” serves as the identity of the machine. This name of the machine is the one through which other machines can access it. For example, one of the machines in this dissertation is serl.cs.colorado.edu. Other machines can access this machine through this name. Moreover, each machine is assumed to have a single ip address. This ip address is stated in the “Machine IP Address” field of the machine model.

This model also lists which application installations are available on it. Applications are deployed through their respective installations. These applications could only be deployed if their respective installations are available on the same machine where the component is present. Due to this requirement a record of application installations for each machine is kept in this model. The application installations are available under a pre-specified directory in each machine. Therefore, only the name of the application is required in this model to deploy the application from installation directory to the working directory.

Like the components the machines are also under variable load. Therefore, the “Current Load” field has the information about the load on a machine in a given time period. This load is calculated in various ways like the processor load or response time request. Other ways to calculate ”average load” are commands such a top, procinfo, uptime and w in unix and its variant operating systems. Some of the unix utilities like sysstat also provide some load information. Because of various ways of getting this load information we have not identified a single way of calculating this load average in this dissertation. However, the value that we have used in this dissertation is the “calc_load” method in unix kernel. Unix commands like top and uptime use this method to find the average load. This load is stated in percentages. Therefore, the load average percentage is stated in this model under the “Current Load” field.

Moreover, information about the current status of total and available memory is also provided. This information is stated in terms of kilobytes(KB). Similarly, the fields stating the total and current hard disk space usage are also available in this model

to track the amount of space left on a machine. This information is also stated in kilobytes(KB). In unix and its variants the hard disk space is calculated through the commands du (in linux) and df (in freebsd).

Platform information which includes operating system and architecture are also part of the machine model. This information is stated in the fields "Operating System" and "Machine Architecture".

This model also states the components installed on the machine under the "Components Installed" field. The components could be Apache, Tomcat and Mysql. The field 'software available' is not used in the experiments of this dissertation. This field is included for handling dependencies that the components have on other software. For example, to run Tomcat requires a version of jdk. As jdk cannot be characterized as component it cannot be under the intercomponent dependencies. Therefore, a seperate field is required to keep track of all the extra software present in one machine. However, since the components we have used in this dissertation have very few software dependencies, we have not used this field. It is only part of this model to handle any future requirements.

Finally, "Machine State" is the field that keeps track of the state of the machine. The state of the machine could either be 'working' (W) or 'failed' (F). As opposed to the components, no 'not working' (N) state is available for the machine. This is beacuse we do not have a way of remotely turning on the machine.Therefore, whether it is state N or F we consider both states as failed (F).

4.4 Failure Modeling

Since our goal in this project is to automate the failure recovery process, it is important to describe the kinds of failure that we are targeting.

4.4.1 Cause of Failures

A system can undergo various types of failure. Each type of failure is manifested by one or more fault. Many categories of faults have been identified such as the categorization by Avizienis [12]. Another characterization was done by Gray and Lindsey after many years of working with computer systems [48]. They have characterized faults into four types: A *Heisenbug* is a fault that alters its characteristics or disappears when it is researched. A common example of such faults is those that are caused by race conditions and are only possible under a certain environment. Recreating the environment may be difficult in these types of faults. A *Bohrbug* on the other hand is a type of fault that does not hide itself or alter its characteristics when researched. A it Schrodingbug is a type of fault that is only possible under certain unusual conditions. Finally, a *Mandlebug* is a type of fault whose behavior is so complex that it looks chaotic.

Out of these four types of fault, in this project, our focus is not to fix the fault but to recover the entire system. Therefore, we will not directly deal with the types of fault but we will deal with the type of failures caused by these faults. Our goal is to recover the system from failure and not to incorporate any kind of fault tolerance in the system.

4.4.2 Types of Failures

Three types of failure occur in computer systems: permanent, intermittent and transient.

4.4.2.1 Permanent

These failures occur by accidentally cutting a wire, power breakdowns and so on. It is easy to reproduce these failures. These failures can cause major disruptions and some part of the system may not be functioning as desired.

4.4.2.2 Intermittent

These are non deterministic failures and appear on occasion. Most of the times these failures are neglected in testing of the system and only appear when the system goes into operation. Therefore, it is hard to predict the extent of damage these failures can bring to the system.

4.4.2.3 Transient

These failures are caused by some inherent fault in the system. However, these failures are corrected by retrying such as restarting software or resending a message. These failures are very common in computer systems.

4.4.3 Behavior of Failed Systems

The types of failures just described cause the system to behave in a certain way. Three types of behavior are possible in systems after a failure.

A *fail-stop* system is one that does not output any data once it has failed. It immediately stops sending any events or messages and does not respond to any messages.

A *byzantine* system is one that does not stop after a failure but instead behaves in a inconsistent way. It may send out wrong information, or respond late to a message etc.

A *fail-fast* system is one that behaves like a Byzantine system for some time but moves into a fail-stop mode after a short period of time.

In this project our focus is to handle systems that have a fail-stop behavior. It does not matter what type of fault or failure has caused this behavior but it is necessary that the system does not perform any operation once it has failed. In other words it just stops doing anything following a failure.

4.4.4 Failures in Our Model

Our model consists of application, component and machine. As stated earlier component and machine are atomic artifacts in our system. Atomic artifact means that they cannot fail partially. Only total failures are possible in these artifacts. Therefore, both component and machine have three states: *working* (W), *not-working* (N) and *failed* (F).

4.4.4.1 Failure of Machine

Machine failure is when a machine fails because of a hardware component failure such as a hard disk or network interface. It could also fail due to software problems at the firmware level. In any case if after many retries a machine cannot be contacted then the machine is considered failed and recovery operation is initiated. Our assumption is that if after several retries a machine is unable to respond then it is considered failed for all practical purposes. Even if a machine appears to be working latter we will not consider it working.

A failure in a machine results in failure of all components of the machine. Therefore, as soon as a machine is considered failed all the components deployed on it are put into a failed state also.

4.4.4.2 Failure of Component

Failure of a component could be a result of machine failures. A component may also fail due to a software fault. If a component cannot be accessed, it is considered failed and put into the failed state in the component model.

Failure of the component has an effect on the application(s) deployed on it. The application is not considered failed but is put into a not-working (N) state. Although the application is also failed, it is moved to a not-working (N) state instead of a failed (F) state. The reason for this difference is that components and machines are fixed. A failed

state (F) means that they cannot be recovered. However, an application can be recovered by deploying it on other working components and machines. Therefore, technically speaking, applications cannot go into an F state until all possible configurations of the application are exhausted.

4.4.4.3 Failure of the Application

As mentioned earlier failures are not possible in applications. However, due to the failures of machines and components an application can lose its functionality. The functionality loss can be divided into two categories: *total functionality loss* and *partial functionality loss*.

Total Functionality Loss: Applications are deployed on one or more components. All the components need to work in order for the application to provide full functionality. However, if all the components where application is deployed fail then the application suffers a total functionality loss. Total functionality loss is also possible if the application is wholly deployed on one component and the component suffers a failure. Because the component is failed, the application also stops working resulting in total functionality loss.

When an application suffers a total functionality loss then a total redeployment of the application is required on other components in the system. The configuration of this redeployment must be same which the application has before the failure. However, if the configuration is not possible then another less strict configuration is applied. The configuration information for each application is available in the application configuration model (ACM).

Partial Functionality Loss: Partial functionality loss is possible when an application is deployed on more than one component. In this case, all components where application is deployed fail but at least one component is still in a working state. In this case, when at least one component hosting a part of the application is working

and at least one component hosting a part of the application fails, we call it partial funtionality loss of the application.

After a partial failure the application needs recovery. This recovery also requires the recovery process to restore any dependencies of the working parts of the application. The goal here is to bring the application back into the configuration where it was before the failure. If for some reasons the original configuration is not possible than a less strict configuration is applied.

Since this chapter is about the modeling aspects of this dissertation, we discuss the modeling of the planning domain in this chapter also. The actual use of this domain in actual planning is discussed in the next chapter.

4.5 Constructing a Domain

AI Planning requires a number of artifacts to create a plan that goes from the initial state to the goal state. A detailed discussion of planning is given in section 5.3. Out of all the artifcats required to plan, the domain is the most important one. A domain capture semantics of the system for planning. A domain is fixed for the life of the system. It consist of actions that are carried out on the system to move it from one state to another. In this dissertation the domain is modeled to bring the system from a failed or partially failed state to a working state.

A domain can be modeled in a number of ways. However, as far as the knowledge of the author and readers of this dissertation, no published work has gathered major ways of constructing a domain. Therefore, to develop a domain one has to look at examples of other already developed domains. Some good examples are available from the "International Planning Competition"[1]

4.5.1 Scope of the Domain

The first step in developing a domain is to decide how much of the semantics of the system must be modeled. Because the semantics of the system can be complex, one has to abstract out the semantics. Moreover, AI planning has its own strenghts and limitations. AI planning cannot solve each and every complexity of failure recovery. However, it can be a great aid in designing systems with automated failure recovery.

In this dissertation our goals are following:

---

[1] International Planning Competition is a biyearly event that occurs with the International Conference on AI Planning & Scheduling. Different planners compete in this planning competition. The domains used in this competition are good examples of domain construction

- The user is able to specify the instructions at a higher level such that he or she does not have to deal with configuration details.
- The resulting plan from the planners is detailed enough so that one can easily translate it into low level configuration directives.

Keeping these two goals in mind we now start developing our domain. Our driving force in developing this domain and subsequent initial and goal states are the models we describe in the previous sections of this chapter. In this section we will mostly concentrate on modeling aspects of the domain. A complete version of the domain we developed in this dissertation is given in Appendix C.

4.5.1.1 Objects

The first thing to be specified in the domain are types of objects. Objects are the physical and logical items on which actions are performed in the planning domain. All types of artifacts in the system that can be acted upon in one way or another are listed as objects in the domain. Artifacts like service, machine, component, connector and application are some kinds of objects present in the domain. These objects can be further subtyped if required. Subtyping is helpful in cases where for instance a component has two types e.g. Apache and Tomcat. This subtyping achieves the same objective as the subtyping concept in programming languages. Again note here that these are types of objects. The exact instance of the object is described in the 'problem file' (discussed later).

4.5.2 Predicates and Functions

Predicates and functions determine the state of an object. Predicates determine the state of an object or a relationship between two or more objects at a particular time. Functions determine some quantified value associated with one or more objects.

4.5.3 Predicates

Predicates determine the state of an object before and after an action is carried out. Figure 4.1 and 4.2 contain the predicates and functions used in the domain we developed for this dissertation. The predicates form a relationship between one one or more objects like if two components are connected the connection relationship is represented as a predicate *(connection-configured ?ap - apache ? ma - machine ?t - tomcat ?ma - machine ?con - connctor)*. In this predicate the first two parameters are Apache instance and a machine where Apache instance is present. The next two parameters are Tomcat and the machine where the instance is present. The last parameter is the name of the connector that is providing the connectivity between the two component. Similarly other predicates also form a relationship between one or more artifacts in the system. For example *(apache-configured ?ap - apache ?ma - machine)*. This predicate shows that an instance of Apache is configured on the machine specified.

Predicates with a single artifact as parameter are also present in the domain. Usually these predicates represent the state of the artifact. For example, the predicate *(machine-failed ?ma - machine)* represents the state of a machine if it has failed.

Another use of predicates is in specifying a goal state. The predicates in figure 4.2 represent a goal state. Note that all predicates here have different set and/or types of parameters. One or more of these predicates are specified as a goal state. This large number of predicates for goal state is because we want to give the user of the domain as much flexibility as possible in specifying a goal state. Since we have an application-driven failure recovery all of the predicates that are specified by a user involve application. These predicates are used to recover an application from failure.

Two kinds of goal states could be specified in a system: an explicit goal state and an implicit goal state. An explicit goal state specifies exactly what state is required including the specification of all the instances of the objects involved. On the other

hand in an implicit configuration all the instances are not specified and the selection of the particular instances is left for the planner.

The different application-ready predicates are divided into explicit predicates and implicit predicates. For example *(application-ready-1 ?app application)* is an implicit predicate. In this predicate no artifacts are specified. The planner has to decide which objects it needs for failure recovery. The *(application-ready-1a ?app application)* and *(application-ready-1b ?app application)* are similar predicates but the way all these predicates work internally is different. We will discuss the differences between the two when we discuss specifying goal state.

All other predicates in the same figure are explicit predicates where at least some information is specified other then merely specifying the application that requires recovery. For example, the predicate *(application-ready-3-with-connectivity ?app - application ?ap - Apache ?t - Tomcat)* states that the application requires recovery and both Apache and Tomcat are included in the configuration. An exact instance of Apache and Tomcat is to be included when specifying this predicate as a goal state. Moreover, because it states connectivity, both the instances of Apache and Tomcat are required to connect with each other before the application can deliver its desired functionality. Other predicates also specify varying degree of information.

As stated earlier an application can have multiple configurations in ACM. Selection of the appropriate predicate is dependent on the inter-component and intra-component dependencies stated in the configuration.

Since just explaining the predicates without expressing the relationship between them is pointless, we leave the rationale behind predicates for the next section where we discuss the action and their modeling.

Predicates related to application
(application-available ?app - application ?ma - machine)
(application-requires-service ?app - application ?s - service)

Predicates related to the machine
(machine-working ?ma - machine)
(machine-failed ?ma - machine)

Predicates related to apache
(virtualhostadded-in-apache ?ap - apache ?app1 - application ?app2 - application)
(import-application-to-apache ?app - application ?ap - apache ?ma - machine)
(application-available-in-apache ?app - application ?ap - apache ?ma - machine)
(apache-working ?ap - apache ?ma - machine)
(apache-has-configuration-file ?ap - apache ?f - file)
(apache-has-installed-module ?ap - apache ?mo - module ?ma - machine)
(apache-has-module-installation ?ap - apache ?mo - module ?ma - machine)
(apache-module-provides-service ?ap - apache ?mo - module ?s - service)
(apache-providing-service ?ap - apache ?ma - machine ?s - service)
(apache-configured ?ap - apache ?ma - machine)
(apache-installation-available ?ap - apache ?ma - machine)
(apache-configuration-file-require-modification ?ap - apache ?ma - machine ?f - file)
(apache-configuration-file-updated ?ap - apache ?ma - machine ?f - file)
(connectivity-available ?ap - apache ?ma1 - machine ?t - tomcat ?ma2 - machine ?con - connector)
(connector-configuration-required ?ap - apache ?ma - machine ?t - tomcat ?ma - machine ?con - connector)
(connection-configured ?ap - apache ?ma - machine ?t - tomcat ?ma - machine ?con - connector)
(apache-reconfigured ?ap - apache ?m - machine)
(virtualhostadded ?c - apache ?a1 - application ?a2 - application)
(apache-module-require-configuration ?ap - apache ?ma - machine ?mo - module)
(connector-available-in-apache ?con - connector ?ap - apache ?ma - machine)

Predicates related to tomcat
(connector-available-in-tomcat ?con - connector ?t - tomcat ?ma - machine)
(application-available-in-tomcat ?app - application ?t - tomcat ?ma - machine)
(tomcat-installation-available ?t - tomcat ?ma - machine)
(import-application-to-tomcat ?app - application ?t - tomcat ?ma - machine)
(tomcat-configured ?t - tomcat ?ma - machine)
(tomcat-working ?t - tomcat ?ma - machine)

Figure 4.1: Predicates used in the Domain

Predicates for specifying the goal state of the application
(application-ready-5 ?app - application ?ap - apache ?s - service ?t - tomcat ?con - connector)
(application-ready-4 ?app - application ?ap - apache ?t - tomcat ?con - connector)
(application-ready-3 ?app - application ?ap - apache ?t - tomcat)
(application-ready-2a ?app - application ?ap - apache)
(application-ready-2a-with-service ?app - application ?ap - apache ?s - service)
(application-ready-2b ?app - application ?t - tomcat)
(application-ready-1 ?app - application)
(application-ready-3-with-connectivity ?app - application ?ap - apache ?t - tomcat)
(application-ready-3-with-service ?app - application ?ap - apache ?t - tomcat ?s - service)
(application-ready-3-with-connectivity-and-service ?app - application ?ap - apache ?t - tomcat ?s - service)
(application-ready-1a ?app - application)
(application-ready-1b ?app - application)
(application-ready-in-apache ?app - application ?ap - apache ?ma - machine)
(application-ready-in-tomcat ?app - application ?t - tomcat ?ma - machine)

Functions
(time-to-start-apache ?ap - apache)
(time-to-restart-apache ?ap - apache)
(time-to-start-tomcat ?t - tomcat)
(time-to-restart-tomcat ?t - tomcat)
(max-machine-load ?ma - machine)
(current-machine-load ?ma - machine)
(apache-load ?ap - apache)
(tomcat-load ?t - tomcat)
(time-to-import-application-in-apache ?app - application)
(time-to-import-application-in-tomcat ?app - application)

Figure 4.2: Predicates used in the Domain

4.5.4 Functions

Functions are specified whenever a numeric value is involved. Some examples of numeric values include cost, time and hard disk space and so on. Such numeric values may need to be calculated during the planning process. Some of these numeric values are related with time such as restart-time of a component or import time of an application. Others are related with memory space or hard disk space and so on. All of the numeric values involved in the domain are stated in terms of functions. A list of functions used in the domain is given in figure 4.2.

4.5.5 Actions

Developing actions for a domain is the most critical aspect of constructing the domain. We have used a bottom-up approach for developing actions in this domain. Appendix C lists all the actions used in the domain.

However, before delving into the details of the actions modeled in the domain, lets take a look at what role an action play in a domain. An action changes the state of the system. For example, in robot motion planning an action may changes the angle of the elbow of a robot being modeled. Another action may move the robot's foot by some unit. To apply an action to the system the system must be already in a certain state. For example an action cannot change the direction of a robotic arm from 90° to 120° if the arm is not already at 90°. Therefore, to perform an action the system must be in a certain state. This pre-requisite state is called the pre-condition or condition for taking the action. that turns it to 120°. For example, the requirement that the robot's arm is at 90° is a pre-condition for performing the action. The result of applying an action is called an effect or post-condition. Therefore, if the pre-condition of the action is that the robot's arm is at 90°, the effect of the action is that it is changed to 120°.

All the actions have pre-conditions and post-conditions. In addition, an action

has time, cost and other values associated with it. All the artifact that are listed as parameters of the predicates in both pre-conditions and post-conditions are listed as parameters of the action. An action can have more than one pre-condition and post-conditon.

In some cases, the pre-condition of an action are post-condition of other actions. For example, before applying the action that changes the robotic arm's angle from 90° to 120°, another action may have changed its angle from 70° to 90°. The post-condition of an action that changes the direction from 70° to 90° is a pre-condition for the action that changes the direction from 90° to 120°.

Therefore, an action is a network of pre-conditons and post-conditions. These pre-conditions and post -conditions are used to encode the semantics of a given system in form of a planning domain. However, not all pre-conditions of an action are post-conditions of other actions. Some pre-conditions are given as facts in the initial state of the system. The initial state contains some pre-conditions of the system that are actually facts of a given system. For example, in the robotic arm example described above, a pre-condition may be specified as if the robot has an arm of length 15 centimeters only then it can be moved from an angle of 90° to 120°. The length of the robotic arm is a fixed value and cannot be changed. Therefore, the length of the robotic arm is specified as a fact in the initial state. Therefore, if the length of the robot's arm is not 15 centimeters, its angle cannot be changed from 90° to 120°.

Actions in our domain are modeled similarly. We have used a bottom-up approach to develop the domain. Therefore, we start from the predicates that specify the goal state of the planning problem. As mentioned earlier the goal state is specified in terms of the *start-system....* predicates. Therefore, we take one of them and see how the actions are developed using a bottom-up approach.

(application-ready-5 ?app ?ap ?s ?t ?con) is a goal predicate. This predicate involves five artifacts. In plain English this goal means that an *application* needs to

be deployed on *Apache* and *Tomcat*. Apache must provide a *service* and both Apache and Tomcat connect using the *connector*. *start-syatem-5* is the action that makes this predicate true. It is given below:

```
(:durative-action start-system-5
:parameters (?ma1 - machine ?ma2 - machine ?app - application ?ap -
apache ?s - service ?t - tomcat ?con - connector)
:duration
(= ?duration 1)
:condition
(and
(at start (connectivity-available ?ap ?ma1 ?t ?ma2 ?con))
(at start (apache-providing-service ?ap ?ma1 ?s))
(at start (application-ready-in-apache ?app ?ap ?ma1))
(at start (tomcat-working ?t ?ma2))
(at start (application-ready-in-tomcat ?app ?t ?ma2))
(at start (apache-working ?ap ?ma1))
(at start (machine-working ?ma1))
(at start (machine-working ?ma2))
(at start (not (machine-failed ?ma2)))
(at start (not (machine-failed ?ma1)))
)
:effect
(and
(at end (application-ready-5 ?app ?ap ?s ?t ?con ))
)
)
```

Now lets look at the pre-conditions of this action. The last four pre-conditions of this action are facts that must be provided in the initial condition. These predicates ensure that the machines where both Apache and Tomcat are deployed are working and are not in a failed state. Because these predicates are provided as facts in the initial condition we will not discuss them here instead we will discuss them in the next chapter where we describe the development of initial and goal states.

The predicate *(apache-providing-service ?ap ?ma1 ?s)* is a special case. It could be a post- condition of an action or it could be a fact given in the initial condition. Remember from our previous discussion that modules provide services. Modules in our

model can be divided into two catagories: modules that are installed and providing their intended service, and modules that are not installed but could be installed to provide a given service. Therefore, if the module is installed with Apache then this predicate is specified in the initial condition. However, if the module is not installed with Apache then it needs to be installed and configured. In this case this can not be a fact. To handle this, we have three actions that install and configure the module in Apache. These actions are only required when the module is not already installed with Apache. These three actions are given below.

```
(:durative-action add-module-to-apache
:parameters (?ma - machine ?ap - apache ?mo - module ?s - service)
:duration
(= ?duration 1)
:condition
(and
(at start (apache-installation-available ?ap ?ma))
(at start (apache-has-module-installation ?ap ?mo ?ma))
(at start (not (apache-has-installed-module ?ap ?mo ?ma )))
(at start (apache-module-provides-service ?ap ?mo ?s))
)
:effect
(and
(at end (apache-module-require-configuration ?ap ?ma ?mo))
)
)
```

```
(:durative-action configure-module-in-apache
:parameters (?ma - machine ?ap - apache ?mo - module ?s - service)
:duration
(= ?duration 1)
:condition
(and
(at start (apache-module-require-configuration ?ap ?ma ?mo))
(at start (apache-module-provides-service ?ap ?mo ?s))
)
:effect
(and
(at end (apache-has-installed-module ?ap ?mo ?ma))
)
```

```
)

    (:durative-action start-apache-with-service
:parameters (?ma - machine ?ap - apache ?app - application ?s - service
?mo - module)
:duration
(= ?duration (time-to-start-apache ?ap))
:condition
(and
(at start (apache-configured ?ap ?ma))
(at start (not (apache-working ?ap ?ma)))
(at start (machine-working ?ma))
(at start (application-available-in-apache ?app ?ap ?ma))
(at start (>(- (max-machine-load ?ma) (current-machine-load ?ma))
(apache-load ?ap) ))
(at start (apache-has-installed-module ?ap ?mo ?ma))
(at start (apache-module-provides-service ?ap ?mo ?s ))
(at start (application-requires-service ?app ?s))
)
:effect
(and
(at end (apache-working ?ap ?ma))
(at end (apache-providing-service ?ap ?ma ?s))
(at end (application-ready-in-apache ?app ?ap ?ma))
(at start (increase (current-machine-load ?ma) (apache-load ?ap)))
)
)
```

The action *start-apache-with-service* has the *(apache-providing-service ?ap ?ma ?s)* predicate as the effect. Therefore, this action can make the predicate true in the pre-condition of *start-system -5*. However, two predicates *apache-has-installed-module ?ap ?mo ?ma)* and *(apache-module-provides-service ?ap ?mo ?s)* are required to be true in order to make this predicate true. The predicate *(apache-module-provides-service ?ap ?mo ?s)* is a fact and it is specified in the initial state. However, the predicate *(apache-has-installed-module ?ap ?mo ?ma)* requires another action to make it true.

The action *configure-module-in-apache* has this predicate as an effect. Therefore, this action will make this predicate true. However, the pre-condition predicate *(apache-*

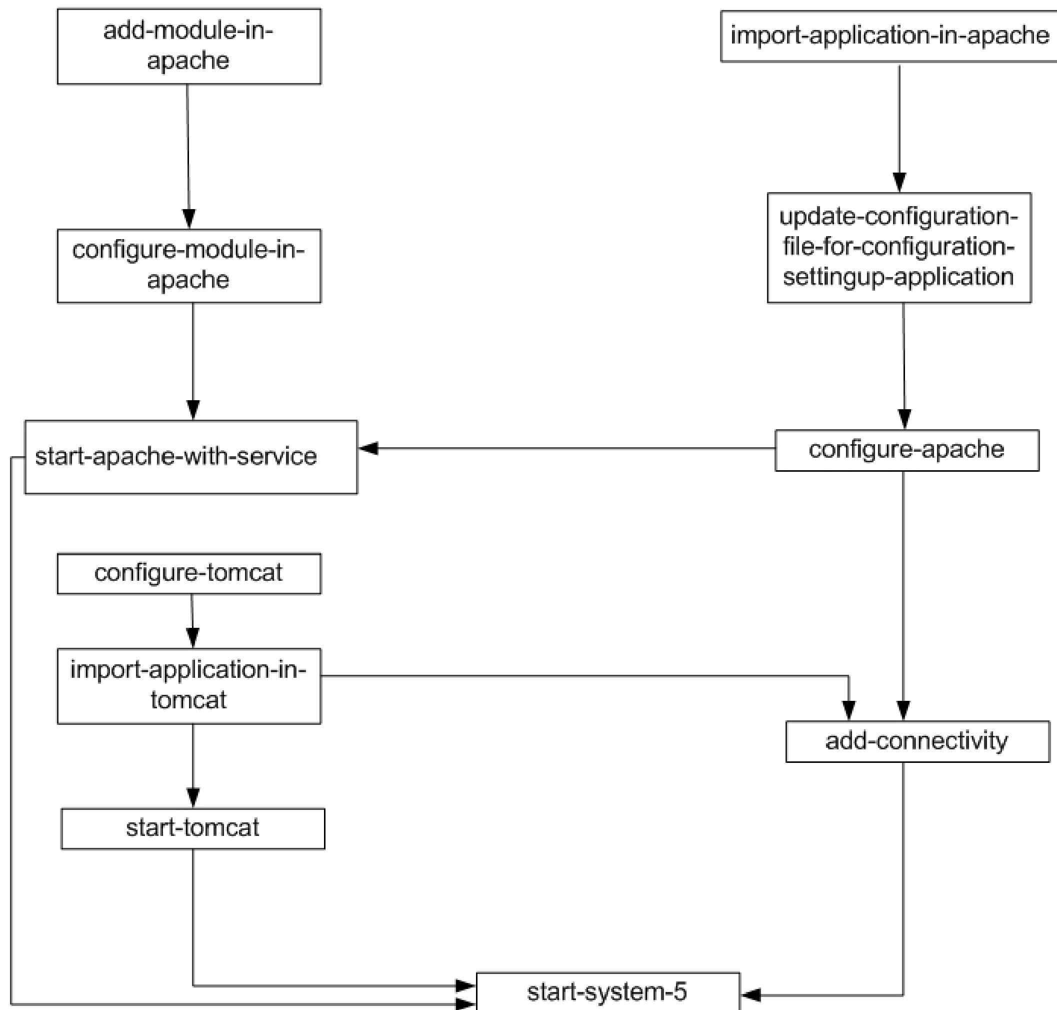


Figure 4.3: Graphical Representation of the Actions for Goal Predicate *(application-ready-5 ? app ?ap ?s ?t ?con)*

module-require-configuration ?ap ?ma ?mo) of this action also needs to be true.

This predicate is the post-conditon or effect of the action *add-module-to-apache*. Therefore, this action will make this predicate true. Note that the action does not have a pre-condition predicate that requires another action to make them true. This is beacuse all pre-conditions of action *add-module-to-apache* are facts and are specified in the initial state.

Therefore, the predicate *(apache-providing-service ?ap ?ma ?s)* requires three actions in the domain to make it useful for the *start-system-5* action. Other predicates in the pre-condition of this action require more or less similar actions. For example, the predicate *(tomcat-working ?ap ?ma)*. This predicate require the following action to make it true.

```
(:durative-action start-tomcat
:parameters (?ma - machine ?t - tomcat ?app - application)
:duration
(= ?duration (time-to-start-tomcat ?t))
:condition
(and
(at start (tomcat-configured ?t ?ma))
(at start (not (tomcat-working ?t ?ma)))
(at start (machine-working ?ma))
(at start (application-available-in-tomcat ?app ?t ?ma))
(at start (¿ (- (max-machine-load ?ma) (current-machine-load ?ma))
(tomcat-load ?t) ))
)
:effect
(and
(at end (tomcat-working ?t ?ma))
(at end (application-ready-in-tomcat ?app ?t ?ma))
(at start (increase (current-machine-load ?ma) (tomcat-load ?t)))
)
)
```

This action will make the predicate *(tomcat-configured ?t ?ma)* true for use in the *start- system-5* action.

A graphical representation of how different actions are dependent with each other

to make the goal predicate *(application-ready-5 ?app ?ap ?s ?t ?con)* true in action *start-sytem-5* is given in figure 4.3. Note that this figure represents only actions that are required in the worst case scenario. If system after a failure is in a partial working condition so some of the predicates already true and are specified in the initial condition.

All the goal predicates have similar representations. Each goal predicate has one action that fulfills the requirement of the goal predicate. However, that one action may have several pre-conditions. Some of those pre- conditions are stated as the initial condition. Others require actions to make them true. Therefore, predicates like *start-system-5* are considered a glue action that force other actions to execute, in order to make some predicates true.

This kind of modeling is similar to human reasoning. Humans also take actions after a failure. We have tried to model the actions of the domain as close to the administrator as possible.

4.5.6 Actions and Interface

One of our goal in this dissertation is to develop actions such that they can be translated directly to interfaces of components. However, not all actions in the domain can be directly translated to the interface. For example the glue actions like *start-component-5* does not have a corresponding interface. However, the actions that are added in the plan due to this glue action have a direct one-to-one transformation to the interface of their respective component. For example, the action *add-module-to-apache* has a similar script/interface in Apache.

The domain that we have developed for this dissertation is given in Appendix C. More details about the usage of the domain is discussed in the next chapter.

Chapter 5

Automating the Failure Recovery Process

Automating the failure recovery process is the main focus of this dissertation. The automation of failure recovery in this dissertation is achieved by using a technique originally developed for control systems and modified for autonomic computing [8, 9]. This technique is called Sense-Analyze-Plan-Execute (SAPE). A sample time-wise failure recovery diagram using this technique is shown in figure 5.1. In this figure each step of failure recovery is given: steps 1-2 make up the 'sense' phase, steps 3-4 make up the 'analyze' phase, steps 5-8 form the 'plan' phase, and finally 9-12 form the 'execute' phase. In this chapter we describe these four phases and their role in automating the failure recovery in distributed systems. We use a sample failure recovery scenario in this chapter to make the discussion concrete.

5.1 Failure Scenario

In this scenario, we have a system with five machines: leone, serl, serl-back, skagen and tigre, three kinds of components: Apache, Tomcat and Mysql, and three applications: sms, rubbos and webcal. This system is shown in figure 5.2.

Rubbos is an application that require Apache, Tomcat and Mysql in a configuration. Therefore, it is working using the Apache on serl, Tomcat on leone and Mysql on tigre. Sms is another application that requires three components. It is working using Apache and Tomcat on skagen and Mysql on tigre. Finally, webcal requires only Apache

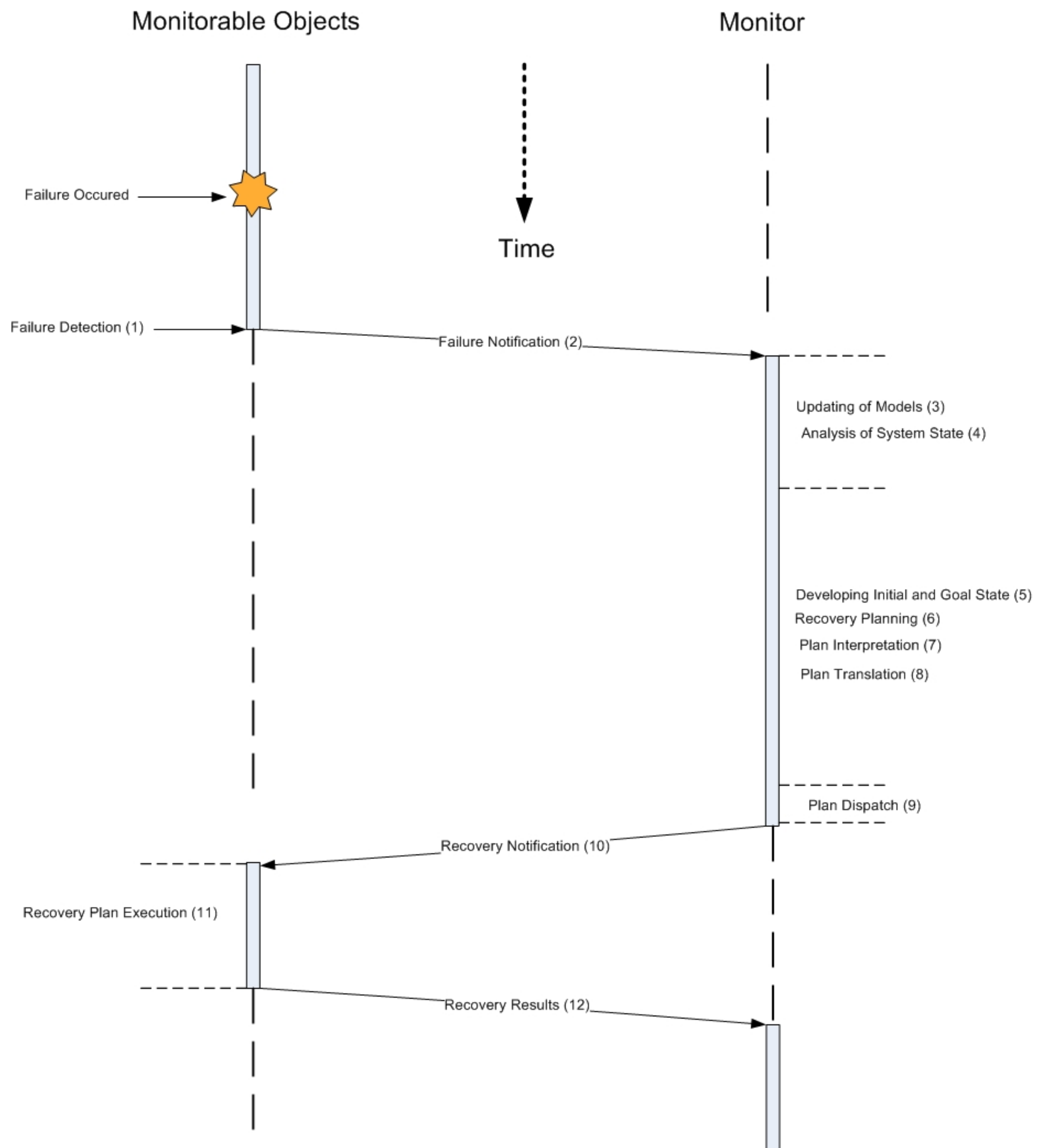


Figure 5.1: Failure Flow Figure

and Mysql. The Apache it is deployed on is on machine serl-back and Mysql is on tigre. We assume that all machines are connected with each other. Moreover, Mysql cannot fail because it has already good failure recovery mechanisms built into it.

The failure that is induced in this system is of machine 'serl'. We will see in this chapter that how the failure recovery process detects this failure and recovers the system. In each section we will describe the theoretical foundations of each phase followed by this example.

5.2 Sense

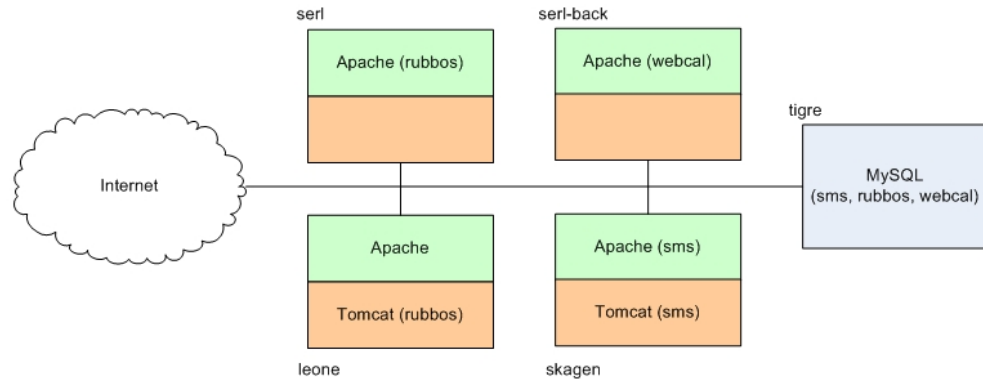
Sense is the phase of failure recovery where a failure is detected. Failure detection is an active area in distributed monitoring and detection. A number of approaches have been developed to detect a failure in distributed systems [41, 59, 94, 49, 85, 66, 35, 52, 38, 70, 44, 25]. However, we have employed an approach proposed by Felber and colleagues [41]. In this section we will first describe their original approach followed by our modification to it.

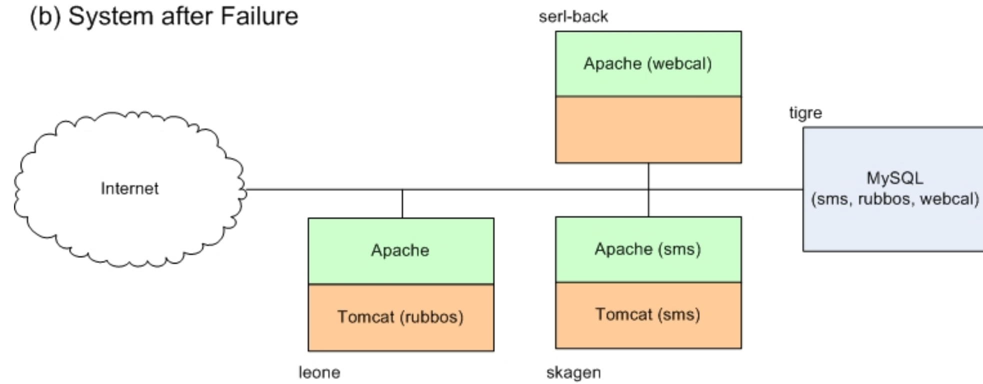
5.2.1 Failure Detection Models

Felber's approach is based on three different detection models: the push model, the pull model and the dual model. All these models have three entities: a monitorable object which is the one being monitored for failure, a monitor for monitoring one or more monitorable objects and detecting any failures, and finally a client to report the results of monitoring of one or more monitorable object.

5.2.1.1 Push Model

A push model is a kind of heartbeat model. A monitorable object sends out a heartbeat message at regular intervals to the monitor. The monitor receives the message and the reception of message indicates that the component is working. If the





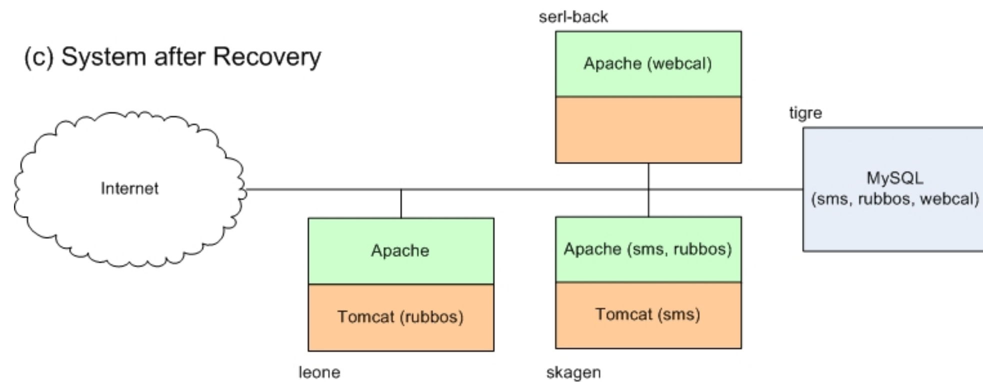


Figure 5.2: System before, during and after Failure Recovery

monitor detects that a message has not arrived within its expected time bounds then the component is considered suspected. A client is only notified when a component becomes suspected. As only one way communication is going on in this model, it is considered an efficient model from a network usage perspective.

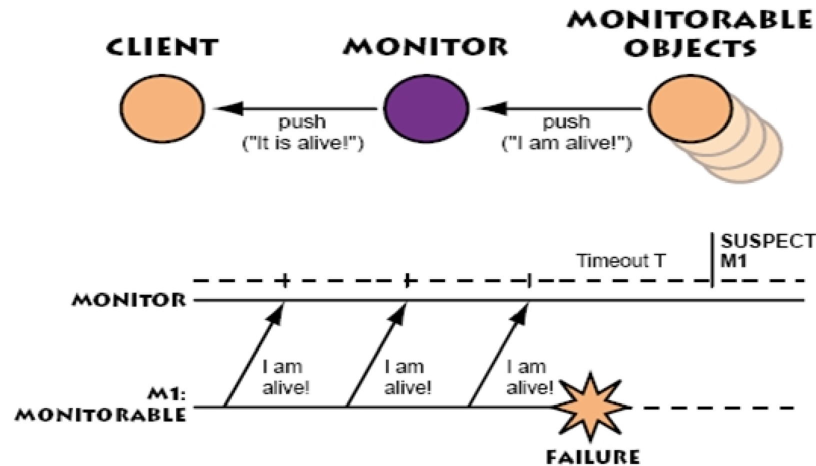


Figure 5.3: Push Model (from Felber et al.)

5.2.1.2 Pull Model

In a pull model the flow of information is opposite to that of the push model. The information is only generated when it is requested. Thus, when a monitor sends out a liveliness request the monitorable object responds to the request. A reply to the liveliness request means that the monitorable object is alive. This model is less efficient as the information flows in two ways. However, this model is easier to implement since the monitorable object does not need to have knowledge of time.

5.2.1.3 Dual Model

The dual model is a mix of the push and pull detection models. It has two phases: in the first phase all the monitorable objects use a push style model. However,

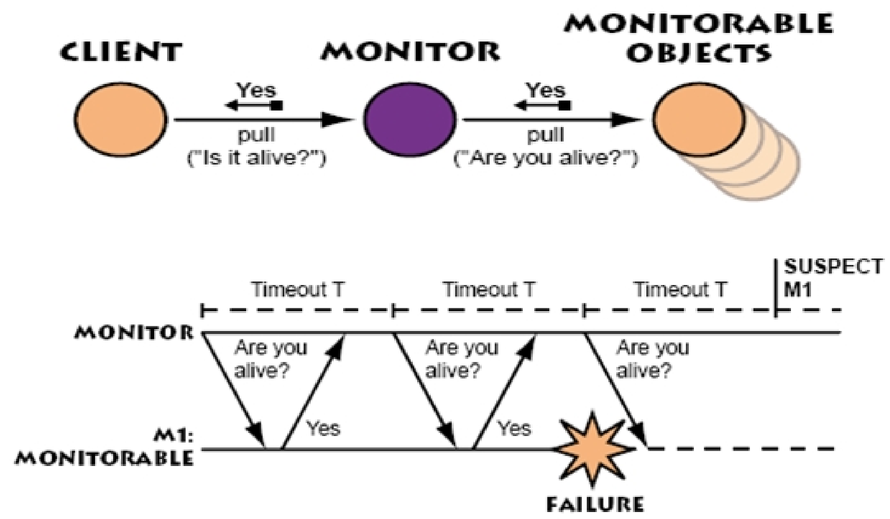


Figure 5.4: Pull Model (from Felber et al.)

as soon as one or more expected messages do not arrive within specified time bounds, the monitor switches to the pull model. The monitor sends out a liveness request to all the monitorable objects whose messages do not arrive within the time bound. If one or more monitorable objects does not respond to the liveness request the monitor puts those objects in the suspected state.

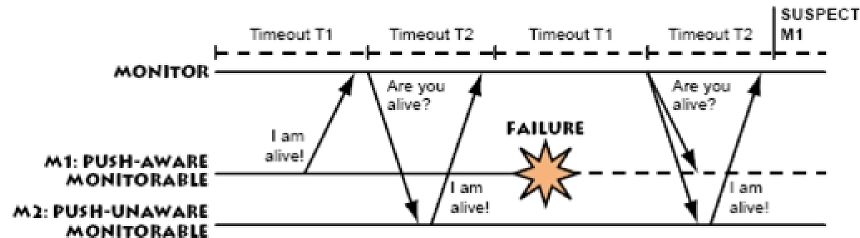


Figure 5.5: Dual Model (from Felber et al.)

5.2.2 Failure Detection in our Model

We have implemented all three models of failure detection in our system. Any of the three methods could be used for failure detection. However, only one model can be used at a given time. Since three types of monitorable objects, i.e. application, component and machine, are involved in our project, we have made some changes to the dual method. Following is the failure detection algorithm that we have employed in our model.

Algorithm for Failure Detection

Each component sends a periodic heartbeat to the monitor. The heartbeat contains a timestamp, machine name and other optional information.

When a heartbeat arrives, the monitor compares the timestamp of a heartbeat to the previous heartbeat received. If the heartbeat has a continuously increasing delta then the monitor puts the component in a SUSPECTED state. A continuously increasing delta shows that the component's performance is deteriorating and the client should

be ready for a possible failure of the component. However, another possibility is that it is just a network blip or other temporary reason and heartbeats after some time start arriving normally. Therefore, if after a certain time the delta is back to its normal limits the component is put back into an ALIVE state.

If monitor does not receive an expected heartbeat from the component then the monitor puts the component in a SUSPECTED state, and performs the following operations:

- Sends an are_you_alive message to the component. If the component responds with a message then the monitor puts the component back in the ALIVE state and continues as normal.
- However, if the component fails to respond to an are_you_alive message then the monitor checks the heartbeats of the other components on the same machine. If the heartbeats of the other components are received normally then the monitor puts the component in a FAILED state.
- However, if the heartbeats of the other components are not received by the component, then the monitor sends an are_you_alive message to the machine. If the machine responds to the message then the components on that machine are put into a SUSPECTED state.
- If the machine does not respond then the machine and all components on it are put in a SUSPECTED state. If there is still no message after a certain time the machine and all the components are put in a FAILED state.

The communication between the monitor and monitorable objects is performed through Siena [22]. Siena is a overlay publish subscribe network. A publisher publish events and a subscriber recieve those events. The relevent information is sent across the network using Siena events. In our system the monitor and monitorable objects both

serve as publisher and subscriber of events in different situations. The monitor acts as a subscriber of events in steps 2 and 12 in figure 5.1 while a monitorable object acts as a publisher of these events. Whereas in events such as 10 the monitor serves as a publisher and a monitorable objects serve as subscribers.

A failure detection in the system depends on the delta of heartbeat and/or ping in the system. For example, if the delta of sending the heartbeat is five seconds and a failure is detected two seconds after a heartbeat, the next heartbeat is not until three seconds after the failure. Therefore, failures detection is dependent on the timeperiod between the heartbeats and/or pings. Since the time period between heartbeats and/or pings is dependent on the system, we will not include this time in our total failure recovery time. Our time calculation starts from the time when a heartbeat and/or ping detects a failure.

To detect a failure and to update the monitor about the state of the monitorable object, two kinds of agent are embedded. These two agents are machine agent and component agent. The machine agent is responsible for the failure detection at machine level, while the component agent is responsible for the failure detection at component level. We do not need a seperate application agent since we assume that application cannot fail by itself. Therefore, only a failure in component or machine can cause disruption to an application.

Both machine agent and component agent respond to the type of failure detection model in use. Using push model both send an event to the monitor after an interval of time specified by the monitor. In case of pull model both responds to the ping event by the monitor. In dual model, a combination of push and pull model is applied. In dual model, very lightweight events are sent to the monitor to keep the network from clogging. These lightweight events only carry identification information of the machines and components. When the monitor recieves the identification information it does not perform any action until it detects a change in the delta of recieving heartbeat messages

or stops recieving these messages at all. In both of these cases it sends out a ping message to both agents. If the agents are working they respond to the ping message with a relatively heavy message. This heavy message contains information about the performance, load, hard disk space and other variables of the system alongwith the identification information of the machine. If one or more agents fail to respond then the monitor put the component or machine of the respective agent(s) in a failed state.

The detection of the failure is mostly the job of the monitor because our assumption is that if a component fails the agent associated with it also fails. Likewise if a macine fails the agent associated with it also fails. Since both kind of agents fail with their respective entities the failure detection is performed through timeout.

In the example shown in figure 5.1 all the machines and components are monitorable objects in figure 5.2. Therefore, the events shown in steps 2,10 and 12 are an abstraction of events from all of machines and components. Now in example 5.2 the machine serl fails at a certain time. Depending on the failure detection algorithm the failure is detected in steps 1-2 of figure 5.1 .

Once one or more failures are detected the application dynamic model (ADM), component model (CM) and machine model (MM) are updated accordingly. The client is notified about the one or more failures in the system. The client in our project is a dependency analyzer which analyzes the effect of the failures on other parts of the system. This analysis is performed in the next phase of our model which is called 'analyze'. The analyze phase serves as an input to the 'plan' phase. Therefore, there is a lot of terminology that is common between these two phases. For this reason, we present a brief description of the Artificially Intelligent Planning (AI Planning) before actually describing these two phases in the recovery procedure.

5.3 Introduction to AI Planning

In traditional AI planning, there are three artifacts required to find a plan: a domain that encodes the semantics of the system, an initial state that describes the state of the system at the present moment, and a goal state that describes the desired state of the system. The domain is fixed and cannot be changed at runtime. However, the initial state and goal states are variable and are determined by the state of the system after a failure. In order to standardize the planning terminology and to exchange and evaluate results, the AI planning community has developed a standardized language for defining the domain, initial state and goal state. This language is called PDDL (Planning Domain and Definition Language) [68]. The AI community has constructed a number of planners [58] that use different heuristics to compute plans. The results from each planner may be somewhat different for a given problem. However, they all use the basic planning paradigm and take as input a domain, an initial state and a goal state.

5.3.1 Planning Inputs

The planner requires a number of inputs. These inputs are divided into three parts: the domain, the initial state, and the goal state. Figure 5.6 gives a description of domain and initial and goal states. Initial and goal states are both specified in a problem file.

The domain is relatively static. It specifies the following items:

- **Types of artifacts:** this consists of all the artifacts that have a role in determining the system state.
- **Predicates/Facts:** the predicates are associated with artifacts (see the section marked predicates in Figure 5.6 An example might be at-machine, which is a predicate that relates a component (or connector) to the machine to which that

component is assigned. The domain actually specifies simple predicates, which are n-ary relations. These can be combined using logical operators into more complex predicates. As with Prolog, instances of these n-ary relations can be asserted as facts, and a state is effectively a set of asserted facts. Predicates are also referred to as constraints.

- **Utilities:** a variety of utility functions can be defined to simplify the specification (see the functions section of Figure 5.6). An example might be local-connection-time, which computes the time to connect a component given that the component and the connector are on the same machine.

- **Actions:** the actions are the steps that can be included in a plan to change the state of the system (see durative-action items in Figure 5.6).

The output plan will consist of a sequence of these actions. An example is start-component, which causes the state of a component to become active. Actions have preconditions (at start in Figure 3) and post-conditions (effects in Figure 5.6). The post-conditions can add, modify, or remove facts from the on-going state that is tracked by the planner during plan construction. The actions are called durative because they have an assigned execution time that is used in calculating the total plan time.

The initial state represents the current state of the system (see the "init" section of Figure ..). This section defines the known artifacts and asserts initial facts about those artifacts. The goal state represents the desired state of our system (see the"goal" section of Figure 5.6). It specifies predicates that represent constraints that must be satisfied in any plan constructed by the planner. The last line of Figure 5.6 defines the metric that is to be used to evaluate the quality of a plan. In this case, the metric is minimal total execution time for the plan.

5.3.2 Explicit and Implicit Configurations

The initial predicates and the goal constraints are integral parts of the configurations. They can be specified in two different ways: implicit and explicit configuration.

Implicit Configuration: The implicit configuration specifies a non-specific predicate about the system that needs to hold after the plan finishes. For example, it can be stated that component A must be connected, but without specifying exactly to what it is connected. This helps the system to specify partial information as a goal. In cases where the system does not have an explicit configuration of the system, it specifies the goal state in terms of the implicit configuration.

Explicit Configuration: In an explicit configuration the artifacts and their configurations are explicitly described as facts in the goal state. For example, it can be stated that component A is connected, and specifically that it is connected to connector B. Explicit configuration information can be specified in a number of ways, depending on the need of the system. An explicit configuration typically requires the exact specification of the related predicates: connected-component and component-is-connected, for example. The former predicate specifies that a specific component A is connected to a specific connector B and, hence, is an explicit configuration statement. The latter predicate specifies only that component A is connected to some (unspecified) connector. If connected-component(A,B) is true, then component-is-connected(A) must also be true.

5.3.3 Planning Output

The output of the planning process is a plan. A typical plan is shown in figure 5.6 The plan is enumerated chronologically. The steps of the plan take the system from the initial state to the goal state. The plan is parallel and some actions could be executed at the same time.

After briefly describing the basic planning mechanism we now move on to the analyze phase.

5.4 Analyze

The analyze phase is initiated when one or more failures are detected in the system. The goal of this phase is to generate an exact picture of the system. Since a system can go into a number of states after a failure, this phase ensures that a complete system specification is available for the recovery process.

A component based system has many dependencies. These dependencies are required for correct execution of a system. Failures usually disrupt these dependencies and an analysis of the system determines the dependencies that are disrupted after a failure.

A dependency analyzer is used to analyze the state of the system following a failure. Using the terminology of Felber et al. [41], the dependency analyzer serves as a client of failure notification and is invoked by the monitor when a failure is detected. In analyze phase two types of analyses are performed: first, the dependency analyzer determines the effects of failure in the system. Following this analysis a complete state of the system is summarized and given to the next phase, 'plan'.

5.4.1 Determining Effects of a Failure

In order to determine the ripple effects of a failure, the dependency analyzer first checks the application dynamic model (ADM) for the current state of each application in the system. Recall from the previous chapter that any application in the system has only one configuration at a specific time. This configuration must be one of those found in the application configuration model (ACM). ACM has information about the working configuration of all applications in the system at any given time.

If a failure is detected in one or more components further tests are carried out to

check the extent of failure. As stated earlier, one of the goals of this dissertation is to recover the parts of the system that are used by the users. Therefore, the first test is to see if an application is deployed on the component. As applications are deployed across one or more component, various cases are possible with respect to the functionality of any application.

In a configuration where the application is deployed on only one component, the failure of the component means a total loss of functionality for the application. Therefore, the whole application needs to be recovered in this case. Similarly, in a configuration where the application is deployed on more than one component and all components fail then the application loses its full functionality. In both of the aforementioned cases the application needs to be totally redeployed. An example of such a configuration is when an application is deployed on both Apache and Tomcat and both of the components fail. In this case, the application loses its full functionality and needs to be redeployed again.

The third case is when an application is deployed on more than one component. If a failure leaves the application in a state where at least one component is working and at least one component has failed then the application loses its partial functionality. In this case only the affected parts of the application require a recovery. For exmaple, if an application is deployed on Apache and Tomcat and Apache fails after a failure in the system, the part of the application deployed on Apache needs to be recovered. However, the part of the application deployed on Tomcat is not required to be recovered.

Failure of a machine results in similar scenarios since machine failures indicate that one or more components on the machine have also failed. Therefore, machine failures result in one or more of the aforementioned failure situations.

5.4.2 Gathering Complete Application Description

When it is determined that an application has suffered total or partial loss of functionality then the dependency analyzer searches the ACM to get the dependency requirements of the application. The dependency analyzer has already determined the state of the application, i.e. total or partial loss of functionality. It matches the state of the application with the configuration information of the application extracted from ACM. The configuration information serves as the reference model (in terms of control theory jargon). The current state of the application is measured against the reference model. This comparision gives information about the components and their dependencies in the application that are affected by the failure.

Various kinds of dependencies exist in the system [55, 39, 61, 66, 65, 26, 4, 33, 93, 31]. However, two types of dependencies are modeled in our system: intercomponent dependencies and intracomponent dependencies.

In some configurations applications are deployed on more than one component. These components are required to communicate with each other. This communication requirement is called an *intercomponent dependency* in our model.

Moreover, as stated previously an application may also require some services from the component. Some services are provided by the component by default and others by installing additional modules. If a service requires the component to install additional modules then this is called an *intracomponent dependency.*

Although we model only these two types of dependencies in our system, other types of dependencies are possible. Some of these dependencies include platforms dependencies, version dependencies, etc. In this dissertation, we are using machines with similar platform i.e. unix and its variants. Moreover, we are using the same versions of the components across all machines. Therefore, it is not required to model these dependencies in this dissertation.

An example of the working of the dependency analyzer follows: If an application is deployed on Apache and Tomcat and Apache fails after a failure in the system, the dependency analyzer checks the configuration information from the ACM and gets the reference model. The reference model gives information about the intercomponent and intracomponent dependency. The intercomponent dependency is the connection with Tomcat and the intracomponent dependency is some kind of service from Apache like ssl_service or php_service. As Tomcat is already in a working state it does not need a recovery. Therefore, recovery requirements in this case are to establish a connection with Tomcat and to install additional modules in Apache, if an add-on service is required by the application.

Now let us see what happens in the analyze phase in the system described in 5.2. In this system, a machine 'serl' fails. A machine failure means that all the components deployed on it also fail. Only one component is deployed on this machine. This component is Apache and it is hosting part of the application Rubbos. Since component Apache has failed the application Rubbos also loses its functionality. However, Rubbos is not only deployed on Apache, it is deployed on two other components, Tomcat and Mysql, also. Therefore, Rubbos has a partial functionality loss.

When it is determined that a failure has occured in the system, the models MM, CM and ADM are updated accordingly. The machine model (MM) is updated to reflect the failure of machine 'serl'. The component model (CM) is updated to reflect the failure of component Apache on serl. The ADM model is updated to show the failure of component Apache on serl.

The goal of the dependency analyzer is to find enough information that the whole system can be recovered to its configuration before the failure. Therefore, the dependency analyzer checks the application dynamic model (ADM) to see which components have failed. It finds out that the component Apache on serl is failed at the moment. It again queries ADM to see what applications are deployed on it. Once it finds that ap-

plication Rubbos is deployed on the failed component, it gets the working configuration number of Rubbos from ADM. Using the name of the application Rubbos and configuration number it queries the application configuration model (ACM) to get the detailed reference configuration information that corresponds to the given configuration number of Rubbos. In the next step it compares the reference configuration information from the ACM to the current configuration information from the ADM. This comparison is performed to find out about the partial or total functionality loss of the application. Depending on the partial or total functionality loss it generates a description of the current state of Rubbos. This description is not in terms of the ACM or ADM models but it is specified in terms of predicates of the failure recovery planning domain.

These predicates are used in the initial state of the problem file. Since the problem file is generated in the next phase 'plan', these predicates are given to the 'plan' phase as input. These predicates are listed in the initial state of the planning problem along with other predicates in the initial state. The initial state used in this example is given in Appendix A. The initial state also includes the predicates that are given as input by the 'analyze' phase.

5.5 Plan

The 'plan' phase is initiated after the completion of 'analyze' phase. In this phase the artifacts of the planning problem, i.e. initial state and goal state, are generated. Together with these artifacts, the domain is given as input to the planner. The planner performs a search using its heuristics and outputs one or more plans. The best plan is selected and translated for failure recovery.

5.5.1 Initial and Goal States

The initial and goal states specify the current and target configurations of the system respectively. The initial state is the configuration of the system where something

has failed. The goal state is the configuration of the system before the failure occurs.

5.5.1.1 Initial State

The first things described in the initial state are the instances of the objects used in the system. These instances include all the machines, components, applications, modules, services and so on. These instances are an actual reprsentation of the physical and logical entities of the system. The types of these objects must correspond to the types mentioned in the domain.

Together with objects the initial state describes all the facts of the system. Any property of the system that is holding in the failed configuration constitutes a fact. All the facts of the system are to be described in the initial state. These facts can be divided into two broad catagories.

Predicates are facts that describe the state of a system or form a relationship between two or more objects of the system. They describe the logical state of the system. These predicates must have their type specified in the domain of the system. The initial state given in Appendix B shows the predicates of the system in our example. These predicates are in a form such as *(application-available sms skagen)*. This particular predicate is of type *(application-available ?app - application ?ma - machine)*. Here the actual instance of an application and a machine are specified. The application is sms and the machine is skagen. Similarly, predicate *(apache-providing-service apache_skagen skagen ssl_service)* is of type *(apache-providing-service ?ap - apache ?ma - machine ?s - service)*. In this predicate the instances are apache_skagen, which is the identifier for the Apache on skagen, skagen is the machine and the add-on service provided by Apache is ssl_service.

An example of a predicate that shows the state of an artifact is predicate *(machine-failed serl)*. This predicate states that the machine serl has failed. Similarly, predicate *(machine-working leone)* shows that machine leone is working.

Functions, on the other hand, describe quantified values of the system. For example, a function *(= (current-machine-load leone) 10)* describes that the current machine load of the machine leone is 10%. Similarly, the function *(= (time-to-restart-apache apache_skagen) 3000)* states that the time to restart Apache on skagen is 3 seconds.

All the predicates and functions describe the current state of the system. These predicates and functions are taken from the three models ADM, MM amd CM. These models capture the state of the system at any given time so the information from these models is accurate up to the last time they were updated.

Apart from predicates and functions generated by these models the predicates given to the 'plan' phase as input from the 'analyze' phase are also listed with the initial condition.

5.5.2 Goal State

Goal state describes the target configuration of the system. We assume that the goal state is specified to recover the system to its original configuration before the failure. However, this assumption is for the current chapter only. In the next chapter we will relax this assumption and consider cases where the original configuration cannot be restored.

A goal state can be specified in two ways: implicit configuration and explicit configuration.

In an **Implicit Configurtion** the goal state is specified at a higher level. This higher level specification is provided to reduce the time of the automated system or human administrator and to shift the task of finding the lower level operations to the planner. Therefore, in our system the implicit configuration is specified whenever one or more applications are to be recovered. The task of dealing with the dependencies, both intercomponent and intracomponent, are left for the planner. In our system, depending

on the types of components involved, three implicit configurations are possible. These are discussed below with the cases when they are used. In all of the cases below Mysql is considered to be part of the configuration unless otherwise stated.

When an application requires both Apache and Tomcat in a particular configuration we use the implicit configuration *(application-ready-1 ?app - application)* as a goal state. The only information required here is the name of the application to be recovered.

In the second case, if the configuration only uses Apache then the implicit configuration *(application-ready-1a ?app - application)* is used as a goal state. In this case also, only the application name is to be specified.

Finally in the third case if the configuration only involves Tomcat, implicit configuration *(application-ready-1b ?app - application)* is used. Also in this case the name of the application is to be provided.

All of the above implicit configuration predicates look similar. However, they have different names and the semantics behind them are different also. In most cases specifying an implicit configuration predicate is enough to recover the application. But there are cases where some restrictions are placed by the administrator of a system on deployment of applications. For example, an application may expect a lot of hits and should be deployed on machine X because machine X is the most powerful machine to handle the application. Moreover, sometimes the administrator wants the components to connect through a specified connector and not through any connector. In the aforementioned cases and similar other cases we need to give the user enough expressive power in the goal state. Therefore, the second type of goal state, called an explicit configuration, is used.

In an **Explicit Configuration** more expressive power is provided to the user. Because the user may require different levels of expressiveness there are numerous predicates to specify an explicit configuration. Some of the explicit configuration predicates

are described in this section.

The predicate *(application-ready-5 ?app - application ?ap - apache ?s - service ?t - tomcat ?con - connector)* is the most detailed predicate in our domain. This predicate expresses the information about the application, the instances of Apache and Tomcat, the service and the connector. Using all this information the planner can only find a plan if constraints specified through predicates are fullfilled.

The predicate *(application-ready-4 ?app - application ?ap - apache ?t - tomcat ?con - connector)* is less expressive than the predicate described above. It does not have a service specification. Otherwise it is the same as the previous one.

The predicate *(application-ready-3 ?app - application ?ap - apache ?t - tomcat)* is further restricted and does not have the specification of connector or service.

Predicates *(application-ready-3-with-connectivity ?app - application ?ap - apache ?t - tomcat)*, *(application-ready-3-with-service ?app - application ?ap - apache ?t - tomcat ?s - service)* and *(application-ready-3-with-connectivity-and-service ?app - application ?ap - apache ?t - tomcat ?s - service)* all are similar but they have different goal specifications in terms of objects.

Finally, predicate *(application-ready-2a ?app - application ?ap - apache)* is for a configuration where only Apache is required in the goal state. Similarly, predicate *(application-ready-2b ?app - application ?t - tomcat)* is where only Tomcat is required in the goal state.

In addition to the initial and goal states a metric is described to control the plan calculation by the planner. The metric could be any function in the domain like apache-start-time or tomcat-restart-time. The metric could either be maximized or minimized. Furthermore, two other metrics are also possible. These metrics are total-time and number of steps. They both can also be maximized or minimized.

All of the above described pieces information which include object instances, initial state, goal state and metric constitutes a 'problem file' in the terminology of AI

planning.

5.5.3 Plan

The problem file and the domain are given as inputs to the planner. The domain is a fixed entity but the problem file can vary with each failure scenario. Along with these items some optional information can also be specified, for example, the maximum time to compute a plan or which particular algorithm should be used to compute the plan e.g. best-first etc.

A number of domain-independent planners have been developed [100, 5, 6, 81, 71, 47, 42, 29, 99]. Domain-independent means that they are not developed for particular domain but a domain can be developed based on a given problem. Moreover, planners have also been developed to solve reconfiguration problems [58]. The planner we are using in this system is called LPG-td [1] [47]. LPG-td uses local search on planning graph as its planning heuristic. This planning heuritic performs really well in the failure recovery domain we have developed.

A planner usually outputs more than one plan to solve a given problem. Two or more plans are only given as output if the latter plan is better than the former one. The quality of the plan is found through the metric specified in the previous section. For example, of the plan metric is to minimize total-time than another plan is not given as output until it has a lower total time than the previous plan. One can specify an upper bound on how many plans a planner should give as output. The minimum number is one for this bound.

Another optional parameter to control the planner is maximum cputime. This is an optional paramter and is not required to be specified when invoking a planner. However, this feature becomes handy when the recovery process is to be performed in a

[1] LPG is being developed at Universita degli Studi di Brescia in Italy. This planner gave one of the best results in the third International Planning Competition (IPC) held in 2004. Also, this planner, compared to other planners, has a really good error detection capability. This capability is very useful for people who are not from the AI world to find errors in their domain and the problem file.

certain time; for example, in some applications there is a 30 second fail-over period. To find the optimum cpu time for a given domain test runs are required to see approximately how much time the planner requires to give a plan. In these test runs usually there is an upper threshold value by which the planner finds a plan and if more time is given to the planner it will not find another plan. Therefore, when this threshold value is found, it should be used as the cputime given to the planner.

When a best plan is selected a plan interpretation is performed. In this step the plan is interpreted according to the architecture of the system. The plan has actions from the domain required to bring the system from the initial state to the goal state. These actions in the plan are for all of the system. However, the system is divided into components, machines and applications. To find the destination of each action, we have included the name of the machine as a paramter in each action. Therefore, all actions in the plan have a machine associated with them.

These actions are then sorted machine-wise. The actions are translated into their respective interfaces. The parameters of the action are also parameters of the interfaces. Therefore, it is one-to-one mapping. When this mapping is performed the actions are ready to be shipped to their respective machines.

The monitor puts these actions into events and publishes the actions for each machine. Each event sent to a machine has the unique name of the machine so that it is only recieved by the correct machine. This dispatch of the event is controlled by an 'execution manager'. Details of the execution manager are discussed in the next section.

Now let us go back to our example and see how a plan is developed for the recovery in the system in figure 5.2. A problem file is given in Appendix B for this particular failure scenario. The plan given out by the planner is given in Appendix B. we shall discuss each action of the plan.

The very first action *(RECONFIGURE-APACHE SKAGEN APACHE_SKAGEN RUBBOS HTTPD)* selects machine skagen for the remaining actions. The compo-

nent that is used on this machine is Apache_skagen and the application is offcourse Rubbos. The file that is required for the configuration changes is the httpd file of Apache_skagen. The next action *(IMPORT-APPLICATION-IN-APACHE SKAGEN APACHE_SKAGEN RUBBOS)* imports the respective application into the working directory of Apache_skagen. In our recovery scenario it is the Rubbos application that is being imported. The next action *UPDATE-CONFIGURATION-FILE-FOR-RECONFIGURATION-ADDVIRTUALHOST SKAGEN APACHE_SKAGEN HTTPD RUBBOS SMS)* is performing a configuration change in the file httpd of Apache_skagen. This configuration change is adding a new virtual host in Apache. Virtual host is a feature of Apache where two or more applications can work in the same instance of Apache and still can be accessed by their own names. This feature of Apache can only be used if there is another application already working. In this plan there is already an application in Apache named sms working in Apache_skagen. Therefore, this action adds both the applications sms and rubbos to share the Apache_skagen using virtual host directive of Apache. The next action *(RESTART-APACHE-WITH-VIRTUALHOST SKAGEN APACHE_SKAGEN RUBBOS SMS)* is a restart command. This restart command is required to implement the virtual host configuration change just made in the configuration file.

Since a part of rubbos is already working on Tomcat_leone, it is not required to be recovered. However, the rubbos on Apache_skagen must be connected to the rubbos on Tomcat_leone. Therefore, the next action *(ADD-CONNECTIVITY SKAGEN LEONE APACHE_SKAGEN TOMCAT_LEONE MOD_PROXY RUBBOS)* connects both parts of the application using the connector 'mod_proxy'. Finally, an action *(START-SYSTEM-1 SKAGEN LEONE RUBBOS APACHE_SKAGEN TOMCAT_LEONE MOD_PROXY SSL_SERVICE LIBPHP4)* is a kind of glue action that we discussed in the previous chapter. This action does not perform any real operation but it is just there to flag that the recovery process is now complete and no further actions are required.

Note that each action also specifies the respective machines where the recovery process is carried out. For the first four actions the machine is skagen since these actions are being performed on components deployed on skagen only. However, in the add-connectivity action leone is also mentioned since rubbos on skagen and leone need to connect with each other to ensure complete recovery of the rubbos application.

This plan is translated into scripts sorted for each machine. The plan interpretation step translates each step of the plan. However, in some situations not all actions are required to be sent to a particular machine. For example, the add-connectivity action uses mod_proxy to connect Apache_skagen with Tomcat_leone. However, in case of connector 'mod_proxy' no change is required in the configuration of tomcat_leone. Thus, this action is not added in the actions of leone.

The translation of the plan to script for reconfiguration that are infact interfaces is shown below.

```
Machine Name: skagen
Actions:
sh /home/arshad/planit/scripts/importApplicationInApache.sh rubbos
sh /home/arshad/planit/scripts/addNameBasedVirtualHost.sh
/home/arshad/apache2/htdocs/sms sms
/home/arshad/apache2/bin/apachectl -k restart
sh /home/arshad/planit/scripts/addConnectivity.sh rubbos leone.cs.colorado.edu
mod_proxy
```

The three interfaces or scripts required to recover rubbos on Apache_skagen are shown here. The first action, reconfigure-apache, is not translated because it is only an indication that the reconfiguration is performed in Apache_skagen. Since these scripts are developed in bash shell, they also have a 'sh' to execute these scripts. The parameters required for these scripts are also shown next to each script.

These actions are dispatched to the machine skagen using an 'execution manager'. We will discuss the execution manager in the next section under the phase 'execute'.

5.6 Execute

Once the script is ready for dispatch to the machines it is handed over to an execution manager. This execution manager, puts the script in a Siena event and adds other identifying information into the event such as the destination machine name and some other attributes. It then publishes the event in the Siena network.

Machines receive the script and save it in a file. This file contains the scripts to be executed on the machine. It executes the file. If all scripts in the file execute properly then the machines send the execution manager a success signal using another Siena event. The execution manager after receiving the event updates the respective models to save the information about the new configuration. This new configuration is used for any further operation in the system.

Now let us see what happens in our example. The execution manager sends the scripts shown in the last section to the machine skagen. When skagen receive the scripts it executes them. If the execution is successful it sends a event back to the execution manager about the success of the execution scripts. When the execution manager recieves the event it updates the models in the system to reflect the new configuration of the system. The system is now working as shown in figure 5.2(c).

This ends the recovery process. However, we have not discussed the case where a script does not work. This is a special case and we will deal it in the next chapter when we discuss the various failure scenarios during failure recovery of the system.

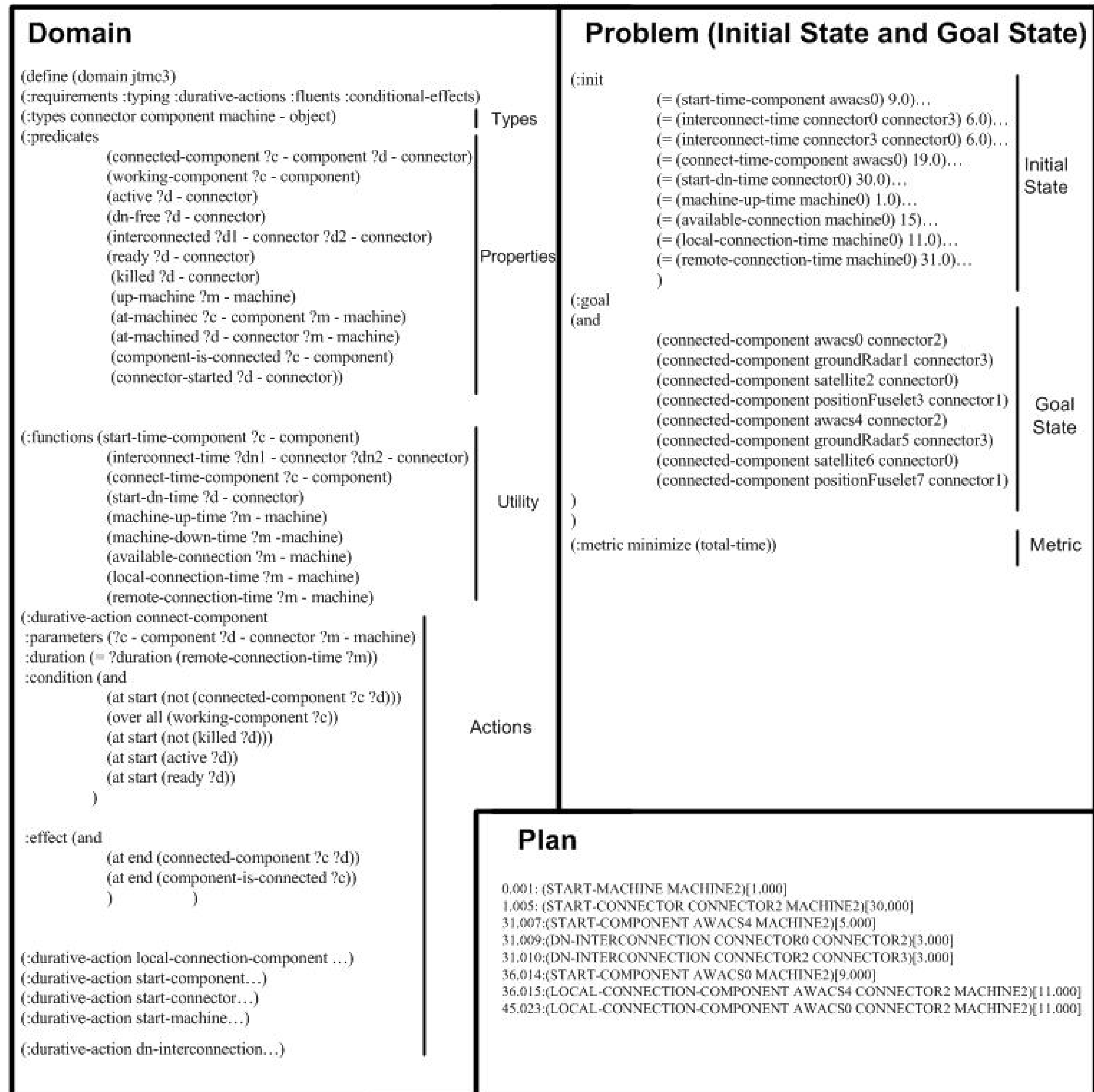


Figure 5.6: Planning Domain, Initial State, Goal State and Plan

Chapter 6

Failures During Failure Recovery

In the previous chapter we discussed the automation of the failure recovery process in distributed systems. Our assumption there was of a perfect recovery process. A perfect recovery means that no failures are possible in or during the recovery process. However, failures in and during the recovery process do occur. Therefore, in this chapter we are going to relax the condition of a perfect recovery process and deal with failures in and during the recovery process. Failures in the recovery process means that the system is unable to recover from a failure. Failures during the recovery process means that more application, component and machine failures are detected in the system while the recovery process is going on. Let us look at each of these failures and see what changes are required in the recovery process to deal with such failures. In addition to these two types of failures, towards the end of the chapter we also discuss a situation when a flawed plan is given by the planner.

6.1 Handling Recovery Failures

In the previous chapter the goal of the failure recovery process was to recover the system to the configuration that it had before the failure. However, in some cases the original configuration may not be possible to attain. When this happens the recovery process has to abort. A human has to intervene and recover the system manually. Since the goal in this dissertation is to automate the recovery process as much as possible, we

will discuss how to deal with such scenarios when the recovery process aborts due to a failure in the recovery process.

The recovery process is divided into four phases: Sense, Analyze, Plan and Execute. In the first phase, 'sense', a failure in the system is detected. Recovery failures are not possible since, during the sense phase, a failure is just detected and no target configuration is chosen as yet for failure recovery. This is because we have chosen a fail-stop model in our system. Moreover, our assumption is that Siena events are guaranteed to be delivered. Therefore, no false-positives or false-negatives are possible in the system. Hence, keeping in view our assumptions, no failures are possible during this phase.

Likewise, in the next phase, 'analyze', the ripple effects of the failure are detected using a dependency analyzer. In this phase, the information is being gathered about the system and no recovery has started yet. Therefore, in this phase also recovery failures are not possible with one exception, the recovery failure that is possible in this phase is when the dependency analyzer is not able to gather correct information about the ripple effects of the failure. However, we assume that this does not happen and the dependency analyzer gathers correct information about the state of the system.

In the next two phases, 'plan' and 'execute', we have to deal with scenarios when an application fails to recover to its original state/configuration. These two phases have two further sub-cases. The first sub-case is when an application is providing some partial functionality and the recovery to the original configuration has failed. In the second sub-case the application is not providing any functionality at all.

We will now discuss the two phases and their sub-cases.

6.1.1 Recovery Failures During 'Plan' Phase

In the plan phase initial and goal states are written down in a problem file and a planner finds a plan for going from the initial state to the goal state. However, it is possible that a plan is not available. This happens because some dependency has not

been met and/or resources are scarce in the system. To deal with this situation, we have added a 'Target Configuration Manager' (TCM) in our architecture. TCM contains knowledge about the existing and other possible configurations of each and every application in the system. In addition, TCM also keeps track of the past configurations of the applications in the system. Therefore, if one configuration is not possible in the system, it should not be used as a goal state in a future failure recovery situation.

Our assumption is that in normal circumstances the system works in its highest configuration. In the highest configuration the application has the highest number of resources and dependencies. Therefore, when a failure first occurs all applications in the system must be working in their highest configuration. Other configuration of applications in the system have less strict requirements. As a rule in this dissertation, the higher a configuration number is, the less resources and/or dependencies it requires.

As discussed earlier two cases are possible when a plan is not available for a given goal state. In the first case an application is still partially providing functionality while in the second case the application has lost its total functionality.

The planner is given a problem file and a domain to find a plan. However, sometimes the planner is unable to find a plan due to various reasons like inadequate resources, etc. When this happens and the application to be recovered has lost its partial functionality, the application is left in a hanging state. This hanging state is not desired because the application cannot be recovered to a full functionality because of the absence of a plan. Therefore, the parts of the application that are working are forced to stop. This is necessary because other configurations of the application may have totally different requirements and dependencies and this working part of the application may not be useful in those configurations. Bringing the working parts to a full stop means that the application is now fully stopped and is in a known state. The act of bringing the application to a full stop is performed by an overall monitor which detects the absence of a plan for a given goal state. The monitor then invokes the TCM

to find another configuration. Completely stopping an application means that now the application needs to be redeployed from scratch. Therefore, this problem turns into a deployment problem in dynamic reconfiguration [8].

TCM selects the next configuration of the application from the ACM. This new configuration is selected as a goal state of the application. Since the new configuration has different dependencies and requirements, the whole initial and goal states are developed again and put into a new problem file. The planner is invoked again and asked to find a plan.

If the planner is able to find a plan then the plan is translated into a script and the recovery process continues. However, if a plan is still not available then TCM is again invoked to find another configuration. TCM then selects another configuration and repeats the the whole process described above. This continues until all the configurations of an application are exhausted. At this point the recovery process has to be halted because no recovery in the system is possible.

In the second sub-case, when an application has totally lost its functionality, the same process as described above, is repeated. However, as the application has already lost its full functionality, there are no running parts of the application. Therefore the steps to stop an application completely are not required. The rest of the recovery process is similar.

6.1.2 Recovery Failures During 'Execute' Phase

When a plan is found by the planner for a given goal state, it is translated and sent to the machines for recovery execution. However, it is possible that a problem occurs in the execution of the recovery script. Problems are possible in the recovery script itself, e.g. in its syntax, etc, or it is possible in the execution of the script, e.g. an application cannot be imported to a directory because the directory is read-only. However, we assume that the scripts do not have any syntax mistakes. Therefore, the

former case is not possible in the system but the latter case is possible.

If a script is sent to the machine agent to execute and it fails then the machine agent must undo all the commands in the script. Most of the commands are control commands like start and stop. They can be easily reversed. However, to undo a configuration change in a configuration file the agent keeps a backup of all configuration files modified. Therefore, when one script does not work the old versions of configuration files are restored. The goal of this excercise by the agent is to ensure that no change is made in the configuration of components and/or applications on a particular machine. After undoing all the changes the monitor sends an event back to the monitor to indicate an unsuccessful recovery.

The monitor keeps track of all machines where it sends recovery scripts. If all the machine agents indicate a sucessful recovery then the monitor updates the models to reflect the changes in the system. However, if even one machine indicates a unsucessful recovery it needs to undo the changes to all the machines.

Therefore, the monitor send undo events to all the machines to undo all the changes they performed recently. When all the machines confirm that they they have performed the undo operation, the monitor initiates a new recovery process.

TCM is invoked to select a new configuration of the applications to be recovered. The recovery process starts again from the 'analyze' phase. The rest of the process is similar to the normal recovery process.

The recovery process just described is for the applications that are not providing any functionality. However, in cases where the application has lost its partial functionality and cannot be recovered, an additional step is required. This additional step is to stop the application completely before invoking the TCM to find another configuration. The rest of the recovery process is similar.

6.2 Handling Failures during Failure Recovery

When a system experiences a failure and the recovery process is being applied onto the system, more failures are also possible in it. These failures can create a problem in it because they change the state of the system which is being used to recover a system. For example, the planner may be using a machine to recover an application but during the recovery process that machine also fails. Now even if the plan is executed onto the system it will not recover the system because the machine is not working anymore. Figure 6.1 is a modified view of figure 5.1 to show the possibilities of further failures during recovery. In this section we will look at ways to deal with such further failures in the system during failure recovery process.

As stated earlier, the theoretical underpinning of our approach is based on the dependencies in the system. The dependencies in the system are specified in the form of a dependency model which is part of the ACM. To make it more concrete we have represented this model in form of a dependency graph in this chapter. Each component of the system is a node in the dependency graph.

6.2.1 Kinds of Dependencies

We have modified the kinds of dependencies to handle failures during failure recovery. There are two kinds of dependencies in the dependency graph: hard and soft. Both of these dependencies are inter-component dependencies.

Hard Dependencies are dependencies representing actual functional dependencies between components without which the dependent component can not provide any real functionality. For example, component A has a hard dependency on component B. If component B fails then component A, although working, can not provide any functionality. An instance of hard dependency in the real world is the dependency of a application server on a servlet engine. This is shown in Figure 6.3 by a solid line in the

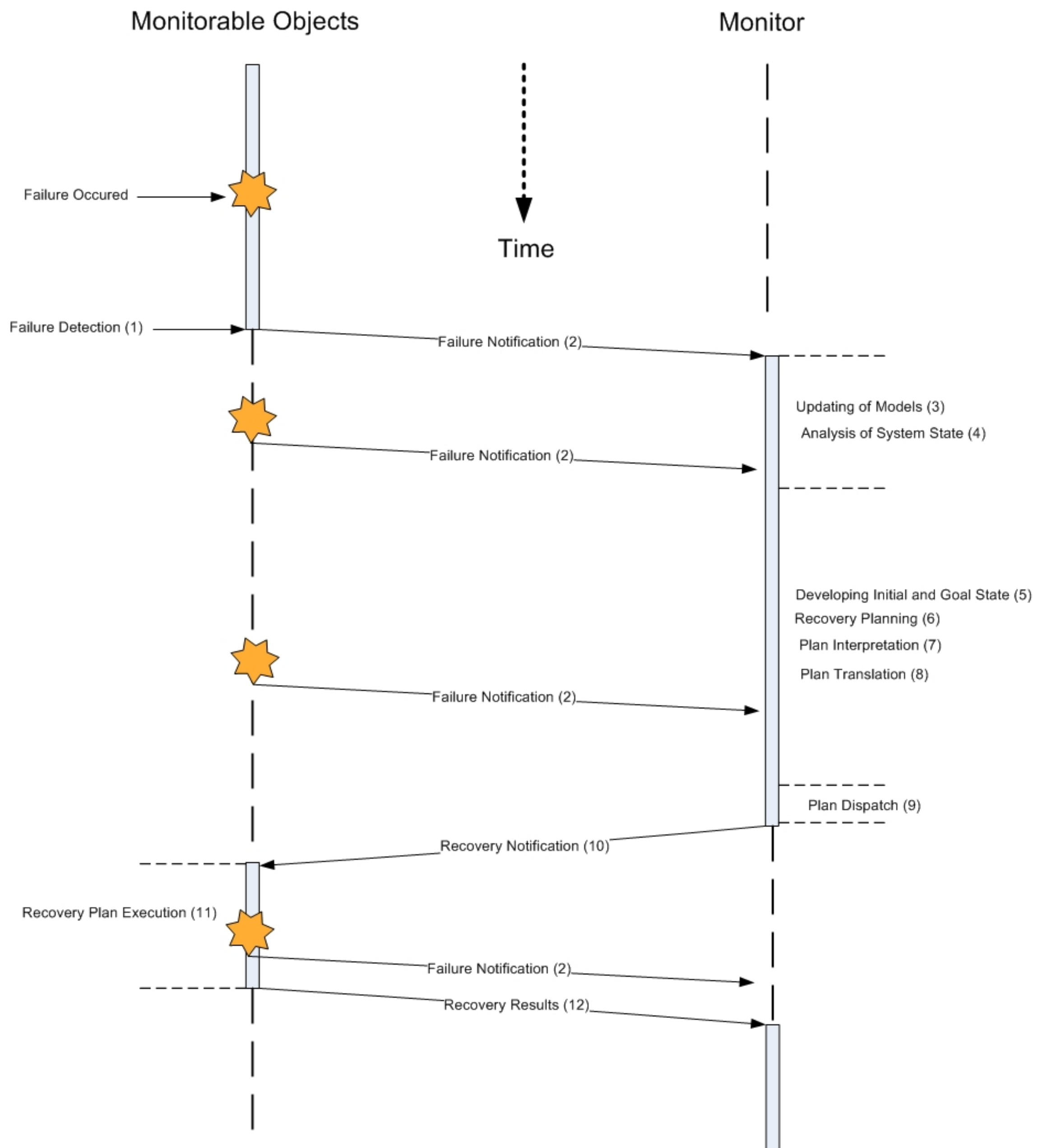


Figure 6.1: Further Failures Possible During Recovery

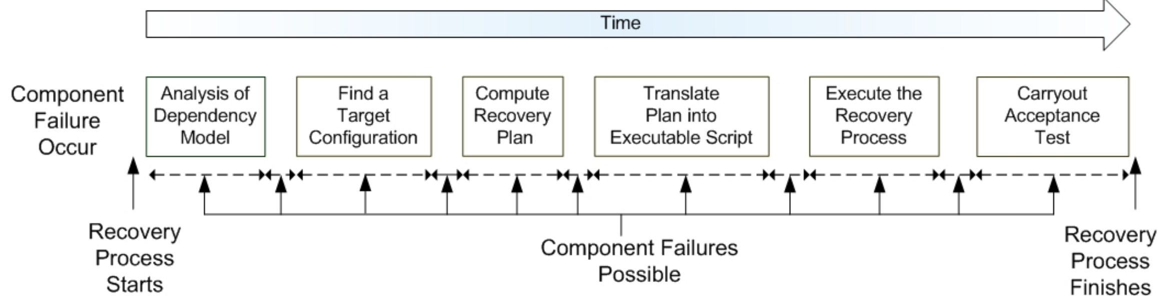


Figure 6.2: Failure detection that results in a decision to either continue or restart the recovery phase.

dependency graph. If the servlet engine fails the application server, although working, can not provide any functionality. This is because servlet engine invokes all the functionality of application server. Without the servlet engine there is no other component that invoke the functionality in the application server.

Soft Dependencies are dependencies representing **use** relationships between components. For example, component A has soft dependency on component C. If component C fails then component A can still provide partial functionality. An instance of soft dependency is a dependency of an http server on a DNS server. This is shown in Figure 6.3 by a dotted line in the dependency graph. If the DNS server fails the http server can be accessed directly by using an IP address instead of full qualified domain address.

6.2.2 Dependency State

The various states that a component can take are working ($\mathcal{W}$), working with no functionality ($\mathcal{N}$), working with reduced functionality ($\mathcal{R}$) and failed ($\mathcal{F}$).

The dependency relationship that determines the state of the component is determined based on whether that component is dependent or antecedent to a failed component and whether the dependency between them is hard or soft. If a component depends on the failed component and has a hard dependency on the failed component

then it is working with no functionality, therefore, it is in state $\mathcal{N}$. However, if a component depends on the component and has a soft dependency on the failed component then it is working with partial or reduced functionality and is in state $\mathcal{R}$. All other components with no dependency link to the failed component are in the working state $\mathcal{W}$.

6.3 Example System

In order to explain our use of dependencies, we give a small example of a real world system. This example is a typical web based system consisting of various servers that we call components in this paper. Figure 6.3 shows the hard and soft dependencies among the different components of the system. Please note that this may not be an actual representation of dependencies among a real application because dependencies are design specific for real world systems.

In our example system there are six components: DNS, HTTP, Servlet, Application server (AS), Database, and SMTP. All these components are assumed to be in a working state. However, their state can change based on their dependency relationship with a failed component.

6.4 Planning

Planning has various sub-phases as shown in Figure 6.2. These phases are invoked sequentially during the recovery process.

The first sub-phase in the planning process is to analyze the dependency graph. The states of the components are determined by analyzing their dependency relationship with the failed component. In order to see how it works lets take a failure scenario from our example and analyze the dependency graph. All the components initially are in state $\mathcal{W}$. Suppose that the component Database fails. Therefore, the database component goes in a failed state $\mathcal{F}$. The Application server (AS) component has a hard

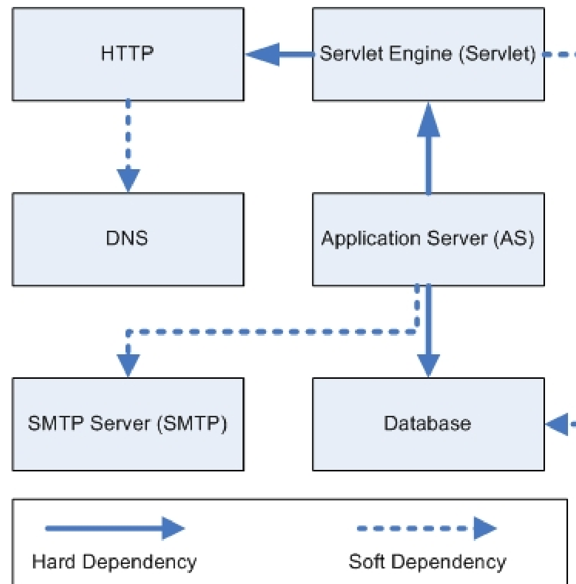


Figure 6.3: Dependency graph of system components.

dependency on the database component so it goes into the state $\mathcal{N}$. The Servlet has a soft dependency on Database so it goes in state $\mathcal{R}$. All other components in the system (SMTP, HTTP and DNS) are in the state $\mathcal{W}$ because failure of the Database does not affect them directly. However, if there is a transitive dependency with a hard edge from any of the components that are currently in a state $\mathcal{N}$ then the transitively dependent component also be in state $\mathcal{N}$. No such dependency is present in our example system.

The next phase is to find a target configuration based on the states of the components in the system. The target configuration may be explicit or implicit. In an explicit target configuration there is a configuration available which gives the details of the component placement, its configuration parameters etc. However, if an explicit configuration is not available then an implicit configuration can be specified. An implicit configuration specifies only the properties that needs to be true at the end of the recovery process. A minimum implicit configuration is "component A must be working" specified in the planning language format.

After the planner executes, it is assumed to produce a plan for getting from the

initial (failed) state to the target configuration. In order to execute the plan, it is converted into an executable script. This script is then given as input to the execute phase.

6.4.1 Handling Failure During Planning

Up to the point where the script is given to the execute phase, all of the planning has been offline, and nothing has actually been done to the failed system. Handling new failures that occur during the planning phase depends on the current state of those components and their relationship to the state of all other components. Components detected as failed can be in any of these states previously $\mathcal{R}$, $\mathcal{N}$ or $\mathcal{W}$. In the following subsection we discuss the failure of the components based on their previous state. and how the recovery system handles these additional failures. Table 6.1 provides a summary of the process.

6.4.1.1 Failure of components in state $\mathcal{N}$

As discussed in the previous section the components in state $\mathcal{N}$ are not providing any functionality. Before starting the planning process, therefore, we treat these components as being in a failed state and are known to our planner.

If a component in state $\mathcal{N}$ reports failure, we are already calculating a plan for its recovery, so we do not need to restart the planning process.

One problem that must be addressed is the restarting of components in $\mathcal{N}$. Such a component may not be providing any functionality, but it may still be running. The solution to this problem is to explicitly stop all the components in state $\mathcal{N}$ after the planning phase finishes. By stopping these components, they truly go into a state $\mathcal{F}$.

It should be noted that this solution, stopping non-functioning components, may actually be unnecessary. It might be the case that the component can be made to function again once all of its antecedents are up and running. We currently do not

take this possibility into consideration because it complicates the planning and allows for better optimization of the resulting plan. An implicit assumption here is that the stopping and starting time of the components is in not significant.

To show how this process works, assume that we start the recovery process of the Database. we assume that because the Application Server (AS) is in a state $\mathcal{N}$ it is also considered to be failed. At the end of the planning phase if AS has not reported a failure, it is explicitly stopped to execute the recovery plan on it.

6.4.1.2 Failure of components in state $\mathcal{R}$

The failure of the components in state $\mathcal{R}$ during the planning process can complicate the plan because these components are presumably providing some functionality, and it may be that some other dependent components are using their functionality. In practice, the handling of components in state $\mathcal{R}$ is pretty straightforward. If a component in state $\mathcal{R}$ fails we have two options. We can either stop the planning process and start it again taking into account the failure of component previously in state $\mathcal{R}$ or we can wait and let the present recovery process finish. Once the present recovery process finishes, we make a second run of the recovery process and recover the newly failed component.

Again in our example: if the Servlet fails during the recovery process then the recovery process of Database (and Application Server) can continue without problem. Once these two recover, then the recovery process is applied to the Servlet. Note that there is a hard dependency of the Application Server on the Servlet. Therefore, unless the Servlet Engine is working, the Application Server can not provide any functionality. In this particular case, then, the planning process has to be stopped and restarted to take the failure of the Servlet into account.

6.4.1.3 Failure of components in state $\mathcal{W}$

Working components can be divided into two categories based on their dependency relationship with the components in states $\mathcal{N}$ or $\mathcal{R}$.

(1) Components that are antecedents of components in state $\mathcal{F}$, $\mathcal{N}$ or $\mathcal{R}$, and

(2) components that are not antecedents of any component in states $\mathcal{F}$, $\mathcal{N}$ or $\mathcal{R}$.

In the first category the failed component is an antecedent of a component in state $\mathcal{F} \wedge \mathcal{N} \wedge \mathcal{R}$. In this case the planning phase must be restarted because there is no point in recovering a dependent component without recovering an antecedent component. Without an antecedent component (assuming a hard dependency), the dependent component will not be able to provide any functionality. Therefore, the antecedent component has to be included in the planning phase to get a better recovery plan.

So if, for example, the Http server fails during the recovery process, it has to be stopped and started again. This is because the Servlet is in state $\mathcal{R}$ and it has a hard dependency on Http. Because Http has failed, the Servlet engine will also be considered as failed. Thus the recovery process has to be restarted while taking into account the failures of Database, AS, Servlet and Http.

In the second category, the present recovery process can continue and finish. After it has completed, the recovery can be planned and executed for the newly failed and totally independent component. For example, if the DNS fails then its can be recovered later because no component in the system has a hard dependency on it.

6.5 Plan Execution

The output of the planning phase is a recovery plan for the system. This plan is translated into an executable script (i.e. a recovery script). The recovery script is executed on the system to bring the components in the system back to the working

state. Again however, additional (or already repaired) components may fail during the execution of the recovery script.

6.5.1 Handling Failure During Plan Execution

We again group the components based on their state during the recovery process. Recall from the previous section that the components in state $\mathcal{N}$ were failed or explicitly stopped. Therefore, we are already recovering them so we will only consider the failure of components in states $\mathcal{R}$ and $\mathcal{W}$.

6.5.1.1 Failure of components in state $\mathcal{R}$

The failure of components in state $\mathcal{R}$ does not cause a significant problem during the recovery process. The components in state $\mathcal{R}$ are dependent so they can be recovered at a later time. Thus the present recovery process can continue without interruption. Once the recovery process finishes, the newly failed component can be recovered by executing the plan-execute phase again on the system.

Rolling back the recovery process in this case can be costly because here the actual recovery is being executed on the system. Therefore, the best alternative is to wait and recover these components later.

6.5.1.2 Failure of components in state $\mathcal{W}$

The failure of the components in state $\mathcal{W}$ can be divided into two categories. The first category is if they are an antecedent of the components being recovered and the second is if they are totally independent.

In the first case the recovery process has to be rolled back. This rolling back is required because without the antecedent component the recovery of the failed components will not actually recover the system. Therefore, rolling back of the recovery process is critical. Once the recovery process is rolled back the plan for recovery again

has to be made by incorporating the newly failed components. After the recovery plan is available the execute phase is carried out on the system.

In the second case the recovery process can continue and finish because the totally independent components are not dependent or antecedent of any component being recovered. Therefore, they can be recovered after the current recovery process finishes.

6.5.1.3 A Flawed Recovery

Another type of failure in the recovery process results from a flawed plan. In this case the recovery process seems to work but the resulting system is not functioning normally or not functioning at all. This may be because the planner produced a flawed plan. Recall that we assume that the recovery process is perfect and it does not make mistakes. The mistake is in the plan that is computed by the planner.

There are two steps involved in this type of failure. First to detect that the system is not working normally. Second, to recover it again.

In order to find out if the system is working normally we use an acceptance test. This acceptance test can be thought to be an online testing of the system; however, it is at a relatively small scale. We assume that the components of the system are already thoroughly tested before deployment. Therefore, we only need to check if the system is properly doing what it is supposed to do after the recovery. In order to achieve this, a set of acceptance tests is carried out on the system. These tests have precomputed results that should be given by a working system. Therefore, the results from the acceptance test from the system are compared against the pre computed expected results. If the results match, it means that the system is restored properly. However, if the results do not match then it implies the system is not recovered properly.

The number of acceptance tests conducted on the system are based on two metrics which cover the whole system functionality. These two tests are yield and harvest of the system components [16].

Yield is the number of tests conducted on the system and how many of them succeeded. If all the tests conducted on the system succeeded then the yield is 100%.

$$Yield = \frac{tests\ completed}{tests\ offered} \tag{6.1}$$

Harvest is the number of components accessed in the system during the testing phase. All the tests conducted on the system must access all the components of the system. When all the components are accessed and the results given out as expected then the harvest is 100%.

$$Harvest = \frac{components\ accessed}{total\ number\ of\ components} \tag{6.2}$$

A 100% yield and 100% harvest means that the system is working properly. However, if the tests do not result into a 100% yield and 100% harvest then there is a problem. This shows that the plan was flawed and we need to re-recover the system.

One of the first steps in this (re-)recovery is to stop all the recovered components and initiate the planning of recovery again. However, in the new initial state given to the planner, it has to be specified that a particular configuration of the system did not work and we need to find a new plan different from the previous plan.

When a new plan is found we repeat the recovery phase with the new plan and test the system again. If the system works as expected then it is considered to be healed. However, if the system does not pass the acceptance test then this process is repeated again until we find a fully recovered system.

Failures during the acceptance test can also occur. However, as the system is recovering these failures can wait until the acceptance test finishes. If the system pass the acceptance test the new failure is planned and executed as a new recovery process. However, if the system fails the recovery process and a new recovery process needs to be carried out on the system then the new failure is included in the previous set of failures. Therefore, the plan that recovers the components include the previously failed components and the newly failed components.

Table 6.1: A summary of what happens to the recovery process when more components fail.

Phase	Sub Phase	Failure of Components in state N	Failure of Components in state R	Failure of Components in state W (antecedent of F, N or R)	Failure of Components in state W (Totally Independent)
Plan	Analysis of Dependency Model	The components are stopped explicitly to bring them to essentially a state F. Therefore, recovery plan already in place	If the component has a hard dependency on components in N then restart Otherwise recover later	Have to restart planning phase	Recover later or in parallel
	Finding a target configuration				
	Computing a Recovery Plan				
	Plan to Script Translation				
Execute	Executing the Recovery Script		Recover later	Recovery process has to be rolled back and restarted	Recover later or in parallel
	Acceptance Test		Recover later	Recover later	Recover later

Chapter 7

Implementation

To evaluate our Sense-Analyze-Plan-Execute (SAPE) methodology we have developed a system called "Recover". Previously we developed another system called Planit to plan deployment and dynamic reconfiguration in distributed systems [8]. Recover is specificially geared towards failure recovery. However, since our approach involves a significant dynamic reconfiguration aspect, we have reused some modules of Planit to plan for dynamic reconfigurations in "Recover".

In this chapter we will discuss the implementation details of all modules of Recover. We have already discussed the details of the working of these modules in the previous chapters. Therefore, our focus in this chapter will be on the architectural and implementation aspects of modules in Recover.

An architecture of Recover is given in figure 7.1. Most of the implementation of Recover is performed using Java 1.5. For communication between remote modules of Recover we have used Siena publish-subscribe network. To execute the configuration in the target system bash shell is used to write and execute reconfiguration scripts.

7.1 Monitor

The monitor is the main module of Recover. It coordinates a number of aspects of a system. A Siena server is used to communicate between all the modules and agents in the system. Therefore, every module and agent needs to register itself with the Siena

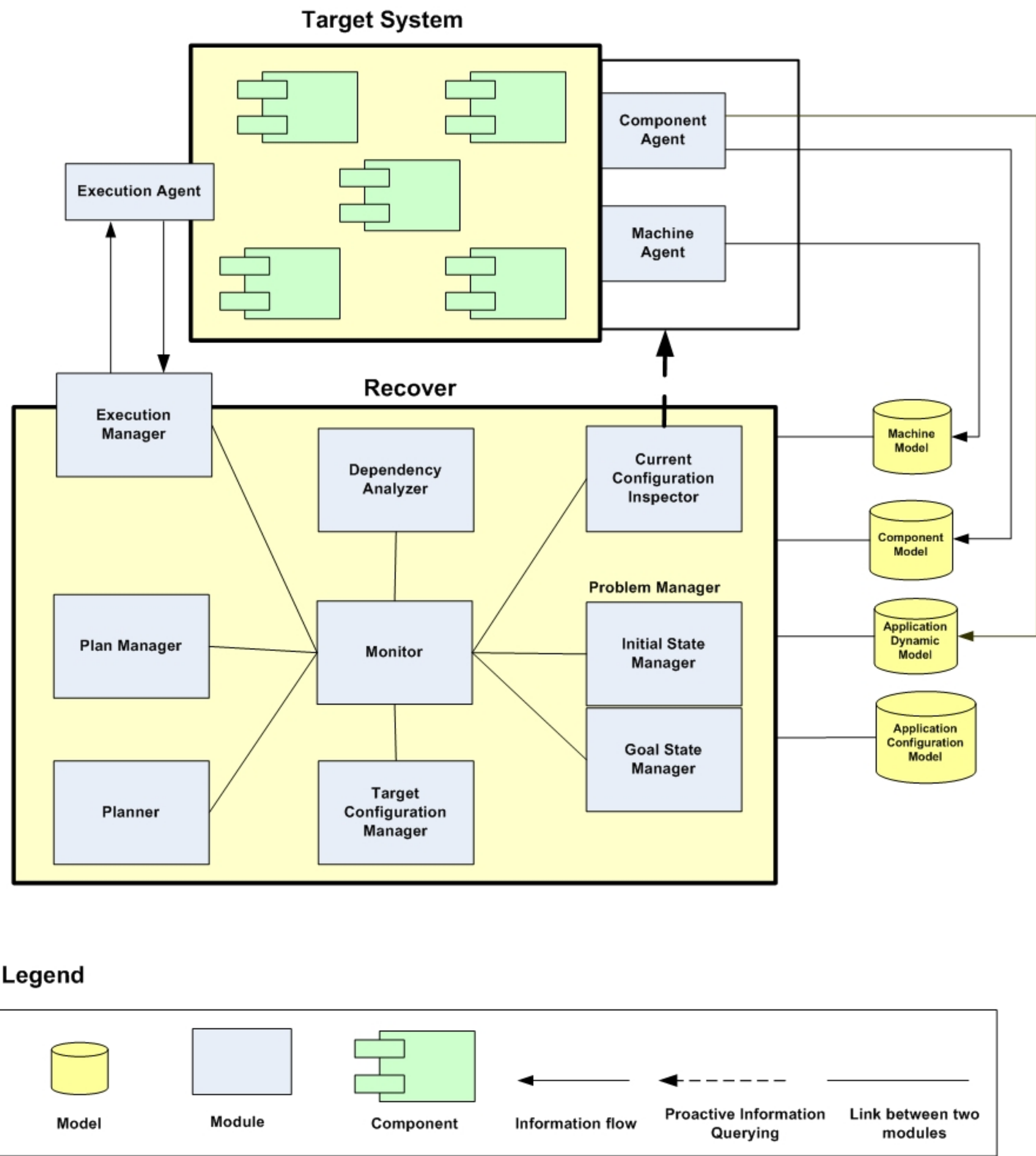


Figure 7.1: A High-Level Architecture of "Recover"

server.

Whenver the system starts, the monitor sends out an identify_yourself_event in the system. This is a general event and all the agents have the ability to receive that event. All the agents that are working in the system and have subscribed to the Siena server receive this event. In response to this event all agents send identification information to the monitor. This identification information includes detailed information about the artifact, i.e. machine or component. This information includes details about the operating system, memory available, etc. When the monitor receive these events it updates the models and adds information about the respective applications, components and machines to their respective models. The monitor then sends an event to all the agents to start the sensing process. A user can select one of the three sensing modes: push, pull or dual. If the user chooses the push model, the monitor only recieves events and updates the models based on the information of those events. Similarly, if the user chooses a pull mode, the monitor sends an are_you_alive message to the agents after a specified time and receives the return event and updates the models accordingly. In the dual mode the monitor works in the push mode first and switches to the pull mode if it detects an unusual behavior in the reception of the events from the agents.

Any of these modes can be used at any given time. The user has to specify a selection. Moreover, the user can also set or change the interval of the events in any of these models. All these selections are performed through the monitor.

The monitor also plays a role in the detection of a failure. The detection of failures is discussed in detail in chapter 5. Once a failure is detected the monitor hands the control to the dependency analyzer to find an exact picture of the application affected by the failure. The dependency analyzer is discussed later in this chapter.

Following this the monitor gives the control to the problem manager for developing the problem file for the planning process. When the problem file is developed, the monitor gives the domain and planner file to the planner to find a plan. The planner

finds one or more plans. The monitor selects the last (and the best) plan given by the planner. It hands over the plan to the plan manager. The plan manager interprets, sorts and translates the plan to the interfaces available in the system.

At this point the monitor gives this plan to the execution manager which sends the parts of the plan to their respective machines. The execution manager reports the success or failure of the execution of the plan back to the monitor. In case of success the monitor updates the models. However, in case of failure it invokes the target configuration manager. The target configuration manager selects the next best configuration and hands it back to the monitor. The monitor gives the new configuration to the problem manager to develop a new problem file. The rest of the process continues as we have discussed before in detail.

All of these coordination efforts make the monitor one of the most important modules in Recover. Furthermore, we assume that the monitor never fails. At this time we have one monitor for the whole system. However, for bigger systems we can use a hierarchical distribution of monitors. However, some changes are required in the architecture to achieve this objective.

In the following sections we describe each module in Recover and what interfaces it uses to interact with other modules of Recover.

7.2 Machine and Component Agents

Both machine and component agents are deployed at remote locations. A machine agent is associated with a machine and a component agent is associated with a component. Only one agent can be associated with each machine and one agent can be associated with each component. Both of these agents work and interact with the monitor. Based on the sensing mode both types of agents respond to events sent by the monitor. In order to send periodic heartbeat both of these agents have a separate thread that sends out a Siena event after every specified time period.

In addition to the information about the component, the component agent also sends information about applications deployed on a component. Both machine agent and component agent use bash scripts and features provided by Java 1.5 to extract information about their respective artifacts. For example, the machine agent gets information about the memory using the unix command "cat //proc//meminfo". Similarly, to find hard disk information the machine agent uses the command "df -h". The component agent also uses bash shell scripts to get information about the component. For example, it checks the status of Apache, i.e. whether it is working or not working using Apache file "httpd.pid". When it finds out that Apache has pid it checks the status of Apache using an http request. If Apache responds to the request, it means that Apache is indeed working. To find about applications deployed on a component the component agent checks the working directory "htdocs" of Apache and the similar directory in Tomcat, i.e. "webapps" to get information about the deployed applications. If an application exists then it must be present in these directories.

7.3 Current Configuration Inspector

The current configuration inspector (CCI) is implemented as a submodule of the monitor. It is used when the dual model is being used as the sensing mode. It explicitly asks the component and machine agents about their status. In response to the event by CCI both component and machine agents send a heavy message and inform about the state and all other information required to get the system models updated. CCI is also used at times to update information about the system configuration. The dynamic values in the system such as applications deployed, hard disk space, memory information, etc. are all updated periodically using CCI. CCI is an optional module and the monitor can work without using it. However, in that case only the push model can be used for sensing.

7.4 Models

As stated in chapter 4, we have four models to describe the state of the system at any given time. These models are Machine Model (MM) to track the information about machines, Component Model (CM) to keep information about components, Application Dynamic Model (ADM) to keep information about an application's configuration and its details at any given time and finally Application Static Model (ACM) to keep static information about the possible configurations of all the applications in the system. Apart from ACM, all models are updated periodically.

These models are updated to reflect the latest state of the machines, components and applications in the system. We have implemented these models using Mysql database. Each of them has a separate table which contains the relevent information. These tables are updated by the monitor only. However, the information contained in them can be accessed by other modules like the problem manager or target configuration manager, etc.

7.5 Dependency Analyzer

The depdendency analyzer is invoked by the monitor when a failure is detected. The monitor does not give any input to the dependency analyzer. As the models are updated before invoking the dependency analyzer, the dependency analyzer checks the models to detect the failure in the system. Moreover, the task of the dependency analyzer is to check all four models and get a complete description of the application. Because of the intercomponent and intracomponent dependencies the dependency analyzer queries the models repeatedly to get the complete state of the application. The actual algorithm which dependency analyzer uses to find the state of a failed application is described in chapter 5. The output of the dependency analyzer are the predicates that describe the intercomponent and intracomponent dependencies in the system. These

predicates are given back, to the monitor in the form of a list. Each item in the list contains one predicate.

7.6 Problem Manager

The problem manager(PM) develops the problem file for planning. The problem file contains instances of objects, initial state, goal state and a metric. PM searches MM to list machines, CM to get components and ADM to get applications in the system. All these instances are specified the format of PDDL in the problem file. Next it writes the initial and goal states of the system.

The initial state is a set of predicates and functions that describes the current state of the system. We have developed templates to list each of the predicates and functions in the system. The objects and values of the predicates and functions are given as input to the respective template. These templates then generate a predicate in PDDL form. All this information about the objects and their values is found through the system models. At this time the templates can only be used for generating the predicates whose types are available in our failure recovery domain. All of these predicates are written as the initial state of the system.

The list of predicates that are given as input by the monitor has the dependency information about applications in the system. These predicates also make up the initial condition. Therefore, this list is also written as part of the initial state by the problem manager.

After writing the initial condition the problem manager uses one of the goal predicates discussed in chapter 5 and writes it as a goal predicate. If the application is to be recovered in a different confguration then the target configuration manager is invoked and it selects a goal state.

Finally, the problem manager writes the metric. The metric in most cases is the “minimize total-time” since our goal is to recover the system in as little time as possible.

After writing all these four parts the problem manager outputs the problem file to the monitor.

7.6.1 Planner

When the monitor receives the problem file, it gives it to the planner, together with the domain, as input. The planner is given a time and a maximum number of plans to find. These two optional parameters are fixed at this time in our system. After many runs we have found that the maximum time to find a plan is less than ten seconds. Therefore, the maximum cputime to find a plan is ten seconds for the planner. Moreover, we also have an upper bound on the number of plans. This number is currently five. This is an arbitrary number and could be increased if desired.

As the time given to the planner is ten seconds, the planner takes approximately ten seconds to find plans for the given initial and goal state. The planner used in this dissertation is LPG-td [47]. LPG-td outputs plans up to the maximum number provided. The monitor checks the list of plans and selects the plan that took the longest amount of time to calculate. This is necessary because the planner does not output another plan until it finds a better plan.

The file containing the best plan is given as input to the plan manager.

7.7 Plan Manager

The plan manager accepts the plan file as input. It scans through the plan and perform various operations on the plan file. It gets rid of all the extra information other then actions. The actions are than sorted machine-wise. When action sorting is complete they are translated into their respective interfaces. The plan manager keeps a dictionary of actions and their respective interfaces. The translation does not involve any more computation because we have one action for each interface.

The plan manager groups the interfaces in a machine-wise list. This list contains

actions that are to be executed on each machine. One list is developed per machine. All these lists are given as output to the monitor.

7.8 Execution Manager

The monitor gives the lists from the plan manager to an execution manager. The execution manager checks each list and extracts the name of the machine. It puts the name of the machine along with the actions in a Siena event. It then broadcasts the events in the Siena network. Each event contains the machine name, actions list and an identifier that tells the execution agent that this reconfiguration is to be performed on the system.

7.9 Execution Agent

An execution agent is associated with each machine. The job of the execution agent is to receive the 'reconfigure event' and execute the reconfiguration. Moreover, it returns the execution results to the execution manager.

When it receives a reconfigure event, it extracts the actions from the event. It puts the actions in a temporary file and exeuctes the file. Before executing scripts, it makes a backup of the configuration files of each component. If all the execution actions are completed sucessfully it sends a Siena event back to the execution manager to signal a success. However, if the execution fails it sends another event to signal a failure.

Execution agent also undoes all the actions that it performed before the failure. Therefore, it reverts the configurations of the system using back up files and restarts them if necessary.

7.10 Target Configuration Manager

The targer configuration manager (TCM) is an auxiliary module of Recover. It is only used if an application cannot be restored into its original configuration before the

failure. If an application cannot be recovered to its original configuration TCM is used to find a new configuration from the ACM. It develops the goal state for the new configuration and hands that goal state to the planner. The planner invokes the dependency analyzer and problem manager and the recovery process continues as normal.

Chapter 8

Evaluation

To test the effectiveness of automated failure recovery techniques, we carried out some experiments using our system 'Recover'. Broadly, these experiments can be divided into three categories: basic experiments, synthetic experiments, and intensive experiments. The first two are quantitative and third one is qualitative. In this chapter we will give details of these experiments and discuss their results.

In both basic and synthetic experiments our goal is to find a recovery plan that restores the failed applications into their configurations before the failure. Therefore, other less strict configurations are not applied to the applications when a plan could not be found to restore them in their original configurations. In the intensive experiments, we will relax this restriction and see how the recovery system performs if an application cannot be restored into its original configuration.

8.1 Basic Experiments

In order to test the effectiveness of the failure recovery process, we have tested it first on a relatively small-scale system. The goal of these experiments is to get a range for the time required for failure recovery. Moreover, since as far as we know this dissertation is the first attempt to use automated planning for failure recovery in distributed systems, it is imperative to test this technique on small-scale systems first, so that we can find any shortcomings or limitations of this approach.

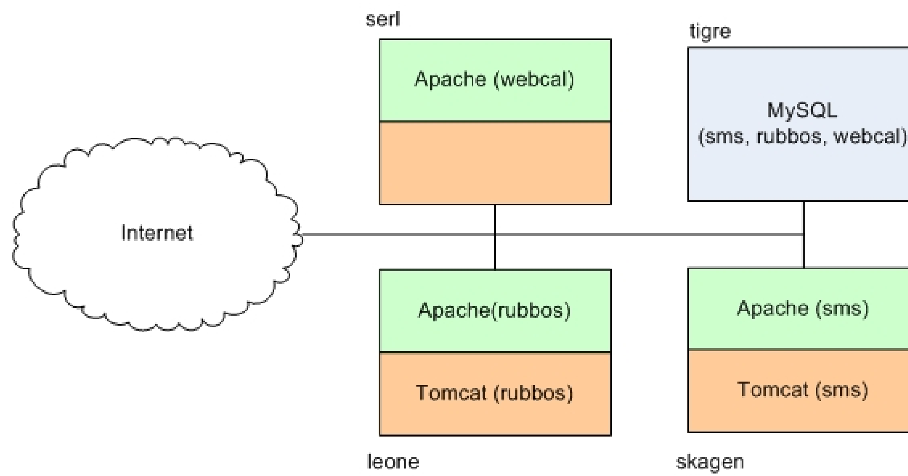
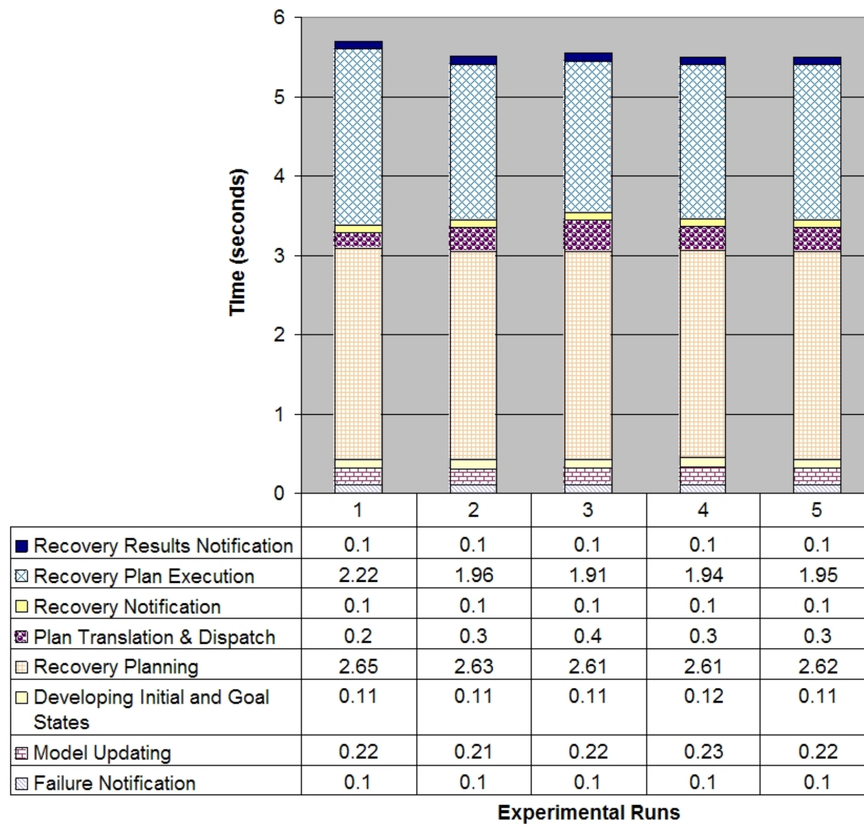
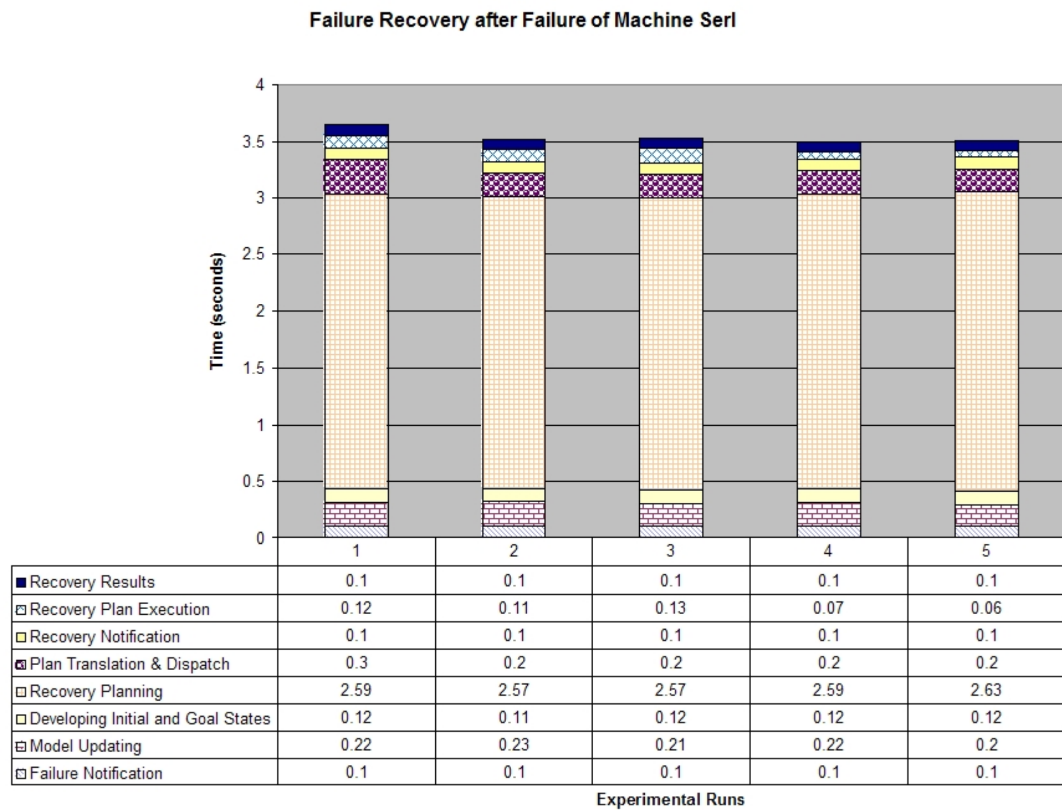


Figure 8.1: Experimental Setup for Basic Experiments



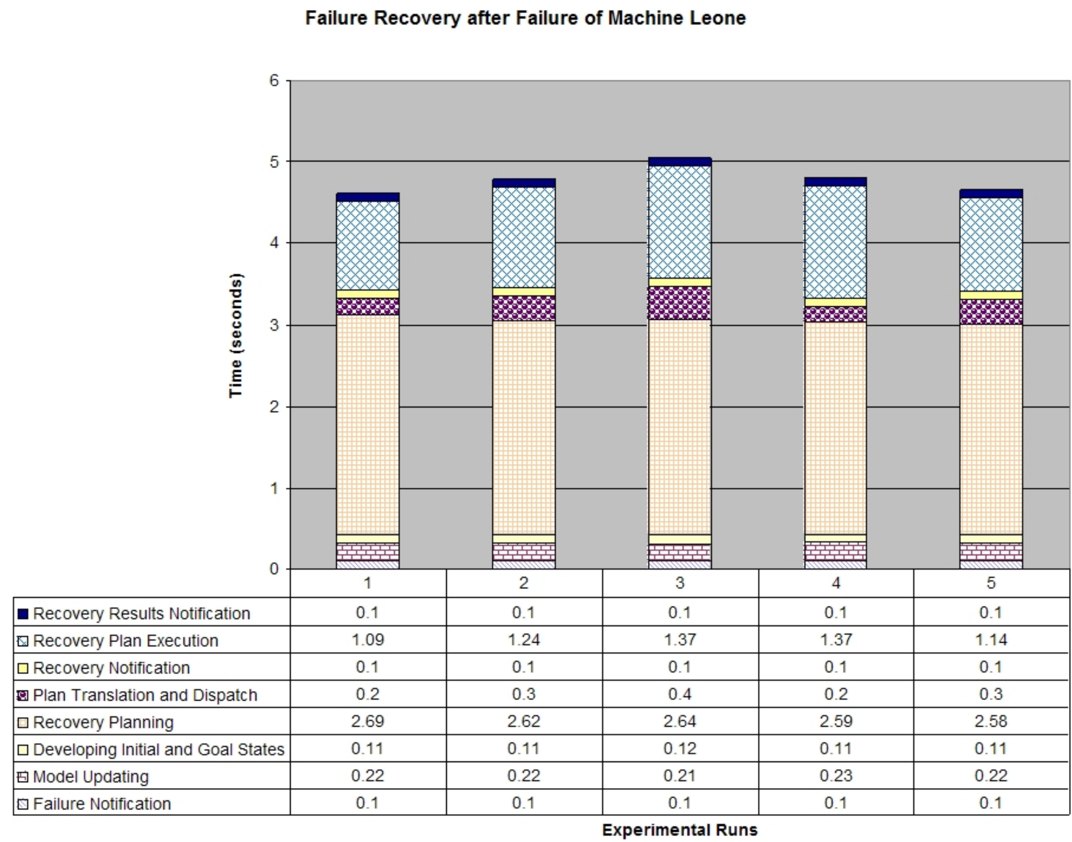
	1	2	3	4	5
Recovery Results Notification	0.1	0.1	0.1	0.1	0.1
Recovery Plan Execution	2.22	1.96	1.91	1.94	1.95
Recovery Notification	0.1	0.1	0.1	0.1	0.1
Plan Translation & Dispatch	0.2	0.3	0.4	0.3	0.3
Recovery Planning	2.65	2.63	2.61	2.61	2.62
Developing Initial and Goal States	0.11	0.11	0.11	0.12	0.11
Model Updating	0.22	0.21	0.22	0.23	0.22
Failure Notification	0.1	0.1	0.1	0.1	0.1

Figure 8.2: Failure Recovery after Failure of Machine Skagen



	1	2	3	4	5
Recovery Results	0.1	0.1	0.1	0.1	0.1
Recovery Plan Execution	0.12	0.11	0.13	0.07	0.06
Recovery Notification	0.1	0.1	0.1	0.1	0.1
Plan Translation & Dispatch	0.3	0.2	0.2	0.2	0.2
Recovery Planning	2.59	2.57	2.57	2.59	2.63
Developing Initial and Goal States	0.12	0.11	0.12	0.12	0.12
Model Updating	0.22	0.23	0.21	0.22	0.2
Failure Notification	0.1	0.1	0.1	0.1	0.1

Figure 8.3: Failure Recovery after Failure of Machine Serl



	1	2	3	4	5
Recovery Results Notification	0.1	0.1	0.1	0.1	0.1
Recovery Plan Execution	1.09	1.24	1.37	1.37	1.14
Recovery Notification	0.1	0.1	0.1	0.1	0.1
Plan Translation and Dispatch	0.2	0.3	0.4	0.2	0.3
Recovery Planning	2.69	2.62	2.64	2.59	2.58
Developing Initial and Goal States	0.11	0.11	0.12	0.11	0.11
Model Updating	0.22	0.22	0.21	0.23	0.22
Failure Notification	0.1	0.1	0.1	0.1	0.1

Figure 8.4: Failure Recovery after Failure of Machine Leone

8.1.1 Experimental Setup

We have developed a small-scale version of an internet service system. In this system three applications are deployed in various configurations. There are four machines and six component instances on which these applications are deployed. An architecture of this system is given in 8.1.

The application Rubbos requires all three components. It is deployed on Apache and Tomcat components of machine Leone. Moreover, it is using the Mysql database on machine Tigre. The application Sms also requires all three components and it is deployed on Apache and Tomcat of machine Skagen. It also uses the Mysql database on Tigre. The third application Webcal requires only two components, which are Apache on Serl and Mysql on Tigre.

Three machines - Leone, Skagen and Serl - have the linux operating system while Tigre is a Windows machine. All of these four machines are powerful server machines with comparable processor speeds and memory.

8.1.2 Induced Failures

In order to test the effectiveness of the failure recovery procedure and to find out about the time required to recover these applications from failure, we have induced machine failures in the system. We tested the recovery of the system by failing one machine at a time except Tigre, since our assumption is that database systems have their own built-in recovery procedures. Note that the machine failures do not involve crashing the machine, but rather stopping all software on it. Our goal is to see how the recovery procedure uses the other resources in the target system to recover from machine failures. This exercise is performed on all three machines, i.e. Skagen, Serl and Leone. We performed each experiment at least five times to get a range required for failure recovery in each case.

8.1.3 Measurements

We have taken various measurements to evaluate the failure recovery procedure. The measurements that we took are the following:

(1) When a failure is detected, the monitor is notified of that failure. Here the time required for this notification is measured. Note that this time does not include the time which is in between failure occurence and failure detection. Instead it is the time between the detection of a failure and the time when the monitor receives a notification about the failure.

(2) The failure notification requires a very small period of time to be delivered to the monitor. Since all the machines in the experimental system are connected through LAN, the time of notification is very small. Therefore, this time cannot be detected precisely because the system clocks of different systems may not be synchronized. Therefore, we have estimated this time and used the estimations in our results.

(3) When a failure is detected, the monitor updates the models CM, MM and ADM. As these models are stored in a Mysql database, various update queries are used to update these models. The overall time required to update these models is measured under *model updating*.

(4) Once the models are updated and a complete picture of the system is in place, the monitor develops a problem file which includes the initial and goal states. The writing of this problem file involves various i/o operations. The total time used by the monitor to write the whole problem file is measured to see the time required by the monitor to develop initial and goal states.

(5) Once the problem file is developed, it is given to the planner along with the domain to find a plan for the given failure recovery scenario. The planner

searches through the problem space and finds a plan. The time required by the planner to find a plan is measured here. This time can also be bounded if the failure has to be recovered in a bounded time. In this case an upper limit to find a plan is given to the planner. This upper limit can be easily found after some test runs. For example, in our case the upper limit to find the best plan is five seconds.

(6) The plan given as output by the planner is interpreted, translated and dispatched to the execution agents of one or more machines. This translation is performed through a lookup table stored in a database. The translation is performed and the actions are packaged machine-wise. The time required to perform this step is measured also.

(7) The time period between the dispatch and reception of the recovery notification at the execution agent's end is measured. Again since we cannot measure this time precisely, we will use an estimate of this time. According to our test runs no notification required more than 100ms to travel from source to destination. Therefore, we will use 100 ms in our estimates.

(8) The execution agent executes the recovery plan. This execution of the recovery plan is measured here. If more than one machine is involved in the exection of a plan then the maximum time used by any machine is used in the measurement. The time required for the recovery plan execution depends on the length of the recovery plan and the time required for these actions. Therefore, this time is very dependent on the target system and its size.

(9) The results of the recovery plan execution, i.e. failure or success, is delivered back to the monitor. The time required to deliver these results is measured here. Again this is an estimation and we will use 100ms as an estimate.

Table 8.1: Recovery time for application Rubbos after its failure on Skagen

Failure of Skagen	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Min	Max	Avg
Failure Notification	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Model Updating	0.22	0.21	0.22	0.23	0.22	0.21	0.23	0.22
Developing Initial and Goal State	0.11	0.11	0.11	0.12	0.11	0.11	0.12	0.112
Recovery Planning	2.65	2.63	2.61	2.61	2.62	2.61	2.65	2.624
Plan Translation and Dispatch	0.2	0.3	0.4	0.3	0.3	0.2	0.4	0.3
Recovery/Notification	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Recovery Plan Execution	2.22	1.96	1.91	1.94	1.95	1.91	2.22	1.996
Recovery Results	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Total	5.7	5.51	5.55	5.5	5.5	5.5	5.7	5.552

Table 8.2: Recovery time for application Rubbos after its failure on Serl

Failure of Serl	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Min	Max	Avg
Failure Notification	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Model Updating	0.22	0.23	0.21	0.22	0.2	0.2	0.23	0.216
Developing /Initial and Goal State	0.12	0.11	0.12	0.12	0.12	0.11	0.12	0.118
Recovery Planning	2.59	2.57	2.57	2.59	2.63	2.57	2.63	2.59
Plan Translation and Dispatch	0.3	0.2	0.2	0.2	0.2	0.2	0.3	0.22
Recovery/Notification	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Recovery Plan Execution	0.12	0.11	0.13	0.07	0.06	0.06	0.13	0.098
Recovery Results	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Total	3.65	3.52	3.53	3.5	3.51	3.5	3.65	3.542

Table 8.3: Recovery time for application Rubbos after its failure on Leone

Failure of Leone	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Min	Max	Avg
Failure Notification	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Model Updating	0.22	0.22	0.21	0.23	0.22	0.21	0.23	0.22
Developing /Initial and Goal State	0.11	0.11	0.12	0.11	0.11	0.11	0.12	0.112
Recovery Planning	2.69	2.62	2.64	2.59	2.58	2.58	2.69	2.624
Plan Translation and Dispatch	0.2	0.3	0.4	0.2	0.3	0.2	0.4	0.28
Recovery/Notification	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Recovery Plan Execution	1.09	1.24	1.37	1.37	1.14	1.09	1.37	1.242
Recovery Results	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Total	4.61	4.79	5.04	4.8	4.65	4.61	5.04	4.778

8.1.4 Experimental Results

The results of our experiments are given in figures 8.2-8.4. Let us discuss one by one the three failure scenarios and their failure recovery results. The times required for various phases of recovery procedure (given in the tables) match the horizontal bars in their respective graphs.

In the case of failure of machine Skagen the results of failure recovery are given in figure 8.2. In the five experiments that we did the recovery took between 5.5 and 5.7 seconds. The two major steps that took most of the recovery time were the planning time and recovery plan execution time. The planning time ranges from 2.61 seconds to 2.65 seconds. The recovery execution time ranges from 1.91 seconds to 2.22 seconds. This time is quite reasonable if we take into account that Apache and/or Tomcat are started and/or restarted during this time which is a relatively time consuming operation. Overall the failure recovery process did very well as it recovered the application Sms that is deployed on Skagen in 5.7 seconds at maximum.

In the second case, machine Serl is failed. The results of the time required for recovery in this case are given in figure 8.3. Here the recovery process completes in a time ranging from 3.5 seconds to 3.65 seconds. Here the range is bit wider than the previous case. The reason for this wide range is the difference in the recovery plan execution time. The recovery plan execution is carried out on one machine in the first three experiments and on a relatively faster machine in the last two experiments. Therefore, he wider range is possible.

In the third case, machine Leone is failed. The recovery results are given in figure 8.4. The range of recovery time ranges from 4.61 seconds to 5.04 seconds. Here also the major contributing factor is the recovery plan execution time. Again because recovery plan execution is performed in different machines using different plans, the overall recovery time also shows a widened range.

The detailed results of basic experiments are given in Tables 8.1-8.3.

8.2 Synthetic Experiments

In the basic experiments, we found that the two factors that contribute most to the length of the failure recovery are planning time required by the planner and recovery plan execution time required by the execution manager. On further investigation we discovered that recovery execution time does not vary with a large number of components and machines. The reason for this is that recovery plan is distributed among one or multiple machines in the system. All execution agents on the machines execute these steps in parallel. The more machines there are in the system, the more possibility if parallel execution is there. However, planning time takes a great deal of time in the recovery process so parallelism is not available while calculating the plan.

Therefore, we decided to carry out further experiments to find the time that the planner requires in large-scale system. To simulate large scale systems is not possible in a laboratory setting because it requires a lot of dedicated resources. Therefore, we developed a simulator that simulates a large scale system and its failure scenarios.

We did two types of experiments using this simulator. The first type of experiments is machine failure experiments. In these experiments we have 20 machines and 20 component instances (10 Apache and 10 Tomcat). One component instance is placed

Table 8.4: Machine Failures: Time to find the First Plan

Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Failures	Min	Max	Avg
61.36	61.21	61.2	61.7	61.2	1	61.21	61.7	61.344
55.64	55.66	55.4	56.78	57.1	2	55.37	57.11	56.112
42.17	42.06	42.5	42.65	42.4	3	42.06	42.65	42.368
31.39	31.61	31.9	31.83	31.3	4	31.32	31.88	31.606
28.4	28.46	28.7	27.71	27.8	5	27.71	28.74	28.228
25.6	25.47	25.7	25.22	25.1	6	25.05	25.72	25.412
22.54	22.2	22.9	22.24	22.2	7	22.2	22.86	22.414
17.05	16.78	16.8	16.85	16.7	8	16.73	17.05	16.844
13.72	12.58	12.9	13.57	12.8	9	12.58	13.72	13.108
11.25	11.37	11.6	12.07	11.7	10	11.25	12.07	11.592

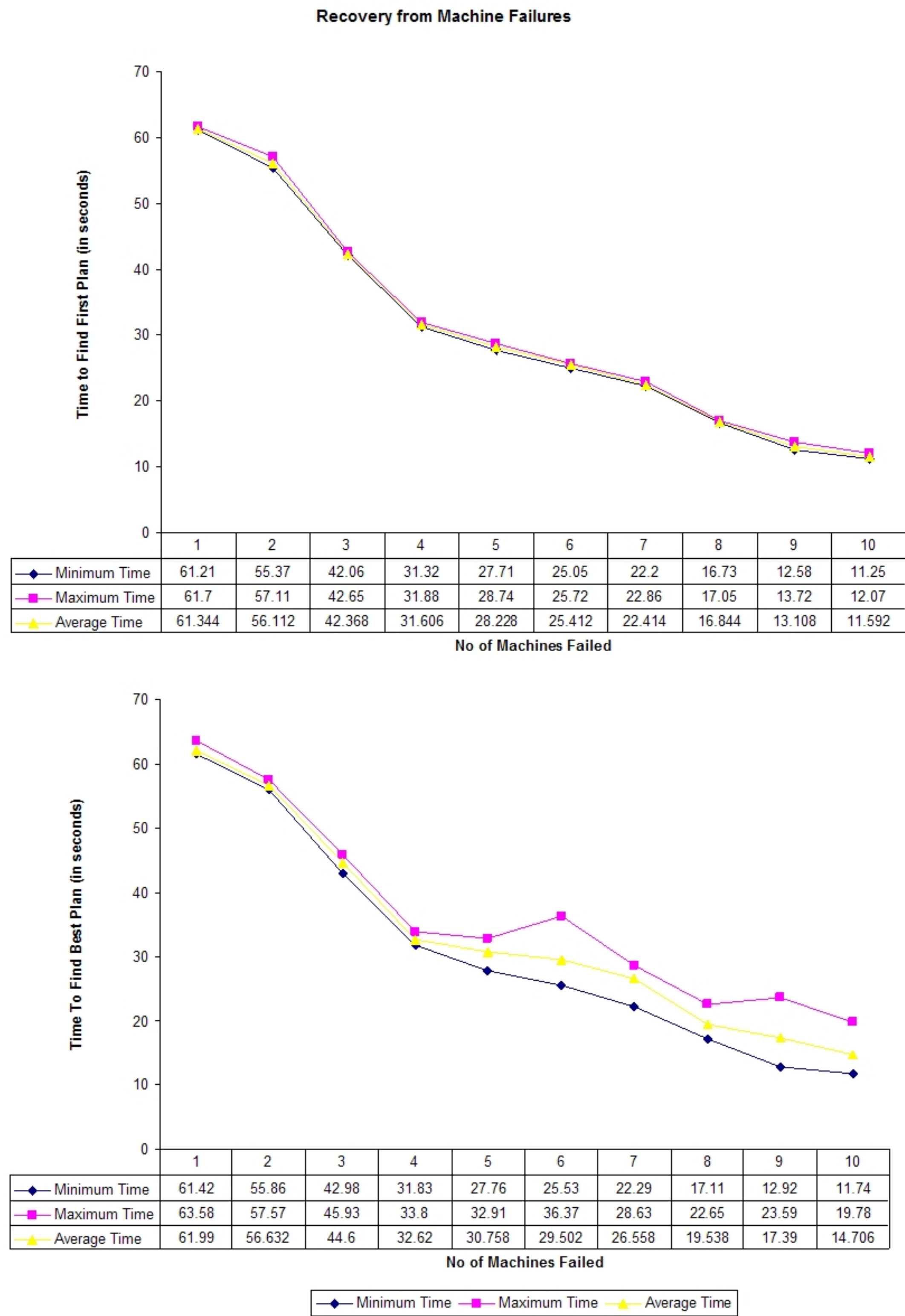


Figure 8.5: Planning time to find a Plan after Machine Failures

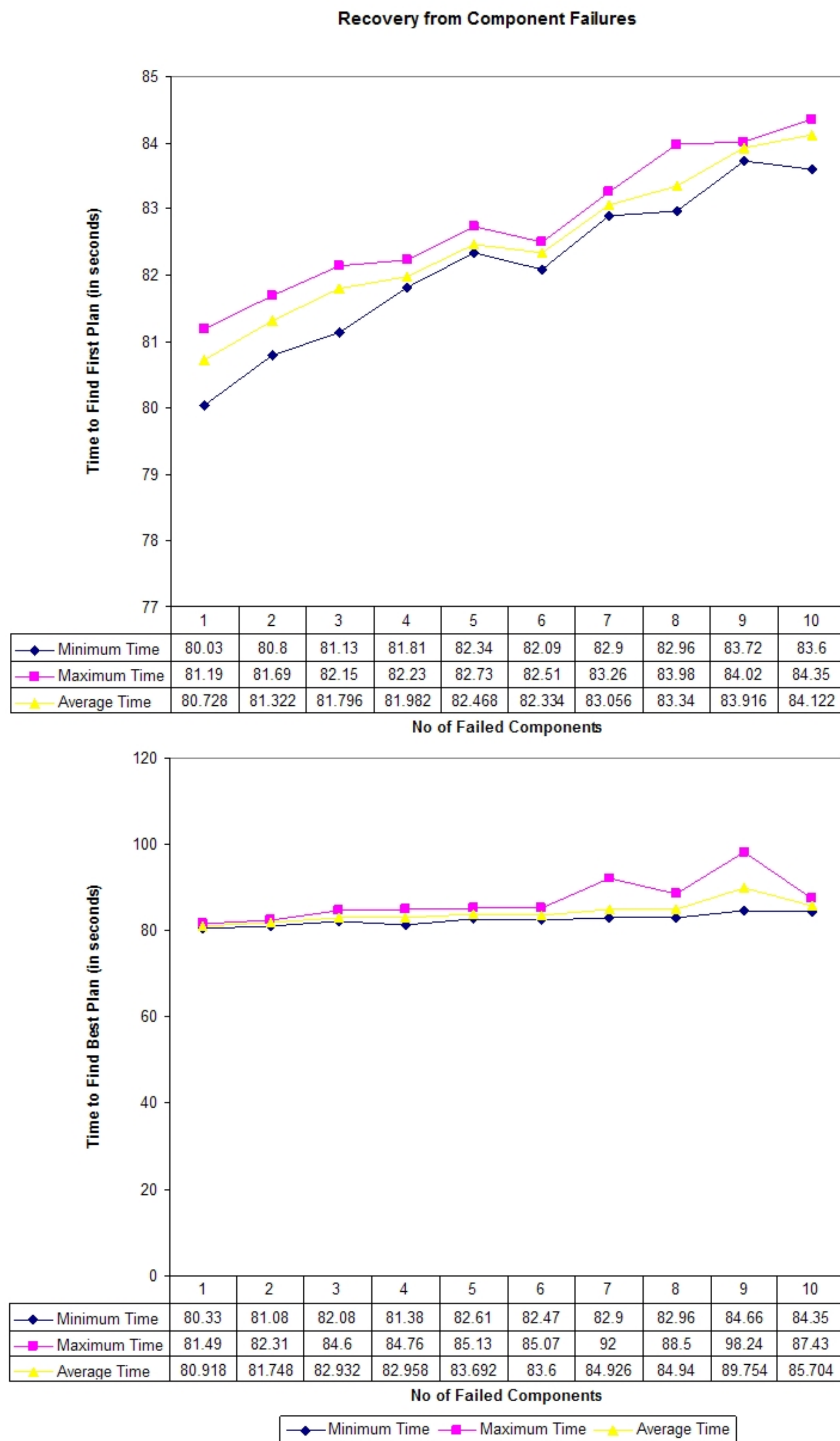


Figure 8.6: Planning time to find a Plan after Component Failures

Table 8.5: Machine Failures: Time to find the Best Plan

Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Failures	Min	Max	Avg
61.5	61.52	61.4	63.58	61.9	1	61.42	63.58	61.99
56.17	55.86	56.3	57.28	57.6	2	55.86	57.57	56.632
45.84	43.21	45.9	45.04	43	3	42.98	45.93	44.6
33.8	31.91	32.3	31.83	33.3	4	31.83	33.8	32.62
31.19	32.91	29.3	27.76	32.6	5	27.76	32.91	30.758
36.37	25.53	30.3	28.84	26.5	6	25.53	36.37	29.502
28.63	26.76	27	28.14	22.3	7	22.29	28.63	26.558
17.31	22.65	22.3	17.11	18.4	8	17.11	22.65	19.538
14.06	12.92	23.6	14.91	21.5	9	12.92	23.59	17.39
19.78	12.79	11.7	12.24	17	10	11.74	19.78	14.706

Table 8.6: Component Failures: Time to find the First Plan

Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Failures	Min	Max	Avg
80.21	81.18	80	81.03	81.2	1	80.03	81.19	80.728
81.43	81.28	80.8	81.41	81.7	2	80.8	81.69	81.322
81.7	82.04	81.1	81.96	82.2	3	81.13	82.15	81.796
82.23	82.08	81.9	81.86	81.8	4	81.81	82.23	81.982
82.36	82.73	82.3	82.46	82.5	5	82.34	82.73	82.468
82.51	82.27	82.3	82.09	82.5	6	82.09	82.51	82.334
82.9	83.26	83.1	82.95	83.1	7	82.9	83.26	83.056
83.98	83.28	83.2	83.29	83	8	82.96	83.98	83.34
83.87	83.95	83.7	84.02	84	9	83.72	84.02	83.916
84.27	84.22	84.2	84.35	83.6	10	83.6	84.35	84.122

Table 8.7: Component Failures: Time to find the Best Plan

Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Failures	Min	Max	Avg
80.33	81.18	80.5	81.12	81.5	1	80.33	81.49	80.918
81.43	82.31	81.1	82.05	81.9	2	81.08	82.31	81.748
82.52	84.6	83.3	82.08	82.2	3	82.08	84.6	82.932
83.61	82.38	84.8	81.38	82.7	4	81.38	84.76	82.958
82.62	83.22	84.9	82.61	85.1	5	82.61	85.13	83.692
82.77	85.06	85.1	82.63	82.5	6	82.47	85.07	83.6
82.9	83.47	92	83.21	83.1	7	82.9	92	84.926
85.85	83.91	83.5	88.5	83	8	82.96	88.5	84.94
88.01	84.66	98.2	84.83	93	9	84.66	98.24	89.754
87.43	84.98	86.3	84.35	85.5	10	84.35	87.43	85.704

on one machine. There are 10 applications deployed on these components. Each application is deployed on one instance of Apache and one instance of Tomcat. Therefore, there are no free resources in the simulated system.

8.2.1 Induced Machine Failures

We induced random failures of machines in the system. In the first case one machine is failed, in the second two machines are failed and so on. We did experiments to test up to ten machine failures. The limit of ten was imposed by the planner. To add credibility to our results we did all these experiments at least five times. During the experiment, we gave an open time to the planner to find a plan.

The results of these experiments are given in figure 8.5. Two graphs are plotted in this figure: The first graph shows the amount of time required to find the first plan and the second graph gives the time to find the best plan. These results are a bit counterintuitive. The more machine failure we induce, the better is the performance of the planner.

The reason for this is that as one of our original assumptions, a machine cannot be restored if it fails. Therefore, the planner is not searching through the facts of the failed machines because it cannot recover anything on them. Consequently, the search graph is smaller, which means the search time is smaller. Section 8.4 provides further discussion.

In the second graph the difference between the time to find the best plan widens among different runs of the experiments. The reason for this is that the fewer resources are available in the system, the more time is required to find a best plan. Less resource availability means that the planner has to search more to find a better plan.

8.2.2 Induced Component Failures

We did similar experiments with components. The difference here is that only component failures are induced in the system and the machines continue to work normally. The same experimental setup is used in these types of experiments. We fail components one by one up to ten components. As there are two types of components in the system i.e. Apache and Tomcat, we fail each type of component turn by turn. Therefore, in our first experiment we have one failed Apache. In the second experiment we have one failed Apache and one failed Tomcat and so on.

The results of these experiments are given in figure 8.6. Table 8.9 gives the corresponding actions and facts of the planning problem with each component failure.[1] In these experiments as more and more components fail, more time is required by the planner to search the graph. Overall the time to find the first plan increases with more component failures. However, as machines are working and the components can be restored (although we are not restoring them) the time to find the best plan is almost the same for the failure of up to six components. As there are less resources in the system remaining, the time to find the best plan increases after more than six failed components.

The details results of both machine and component failures are given in Tables 8.4-8.7.

8.2.3 Computational Complexity Issues

Detailed complexity analyses for planners are quite difficult and have been relatively rare. Some studies have been conducted to assess the complexity of specific domains [53], but we were unable to find any domain-independent analyses.

The most important result in planning complexity comes from a paper by Tom Bylander [19]. In that paper, he shows the rather surprising result that the complexity

[1] The last value in this table is an approximation

Table 8.8: Machine Failures: Corresponding Actions and Facts from the Planner LPG

Number of Machine Failures	Actions	Facts
1	27200	4868
2	25450	4728
3	21920	4188
4	18590	3677
5	17220	3564
6	14280	3092
7	13100	2994
8	11920	2896
9	9560	2478
10	7400	2084

Table 8.9: Component Failures: Corresponding Actions and Facts from the Planner LPG

No of Component Failures	Actions	Facts
1	31140	5446
2	31160	5450
3	31180	5454
4	31200	5458
5	31220	5462
6	31240	5466
7	31240	5468
8	31260	5472
9	31300	5476
10	31320	5480

of a planner is not dependent on the algorithm it uses. It is instead dependent on the planning domain combined with the initial state and the goal state. Therefore, the complexity of a domain is dependent on factors like the number of objects in the system, pre-conditions, and post conditions. These factors change with each domain and planning problem (i.e. initial state and goal state).

Assuming that Bylander is correct, then the proper way to analyze the complexity of a planning system is to examine the domain to which it is being applied. To this end, we measured the number of actions and facts (which includes the pre- and post-conditions) for each of our synthetic experiments described in section 8.2.

Table 8.8-8.9 gives the corresponding actions and facts for each experimental run. They shed more light on our claim that the search space is being reduced. It is clear that the number of actions and facts are going down with each machine failure. Therefore, each experiment has a smaller number of facts and consequently the search tree is smaller. Therefore, the search time with more machines is decreasing in both graphs.

Additional domain measurements would be desirable. Specifically, we would like to see how the actions and facts scale with the addition of more artifacts (machines and components). Unfortunately, our planner is largely a black box, and we cannot obtain these kinds of measurements at the moment. This must remain in our list for future work when we may be able to switch to a different and more transparent planner.

8.3 Intensive Experiments

We performed intensive experiments to test two features of our failure recovery system 'Recover': (1) The capability to handle further failures in the system, (2) The recovery of applications to less strict configurations if the original configuration is not available. The spectrum of possibilities for this kind of testing is very wide. Therefore, we present one test run of one of our experiment we performed to test these two

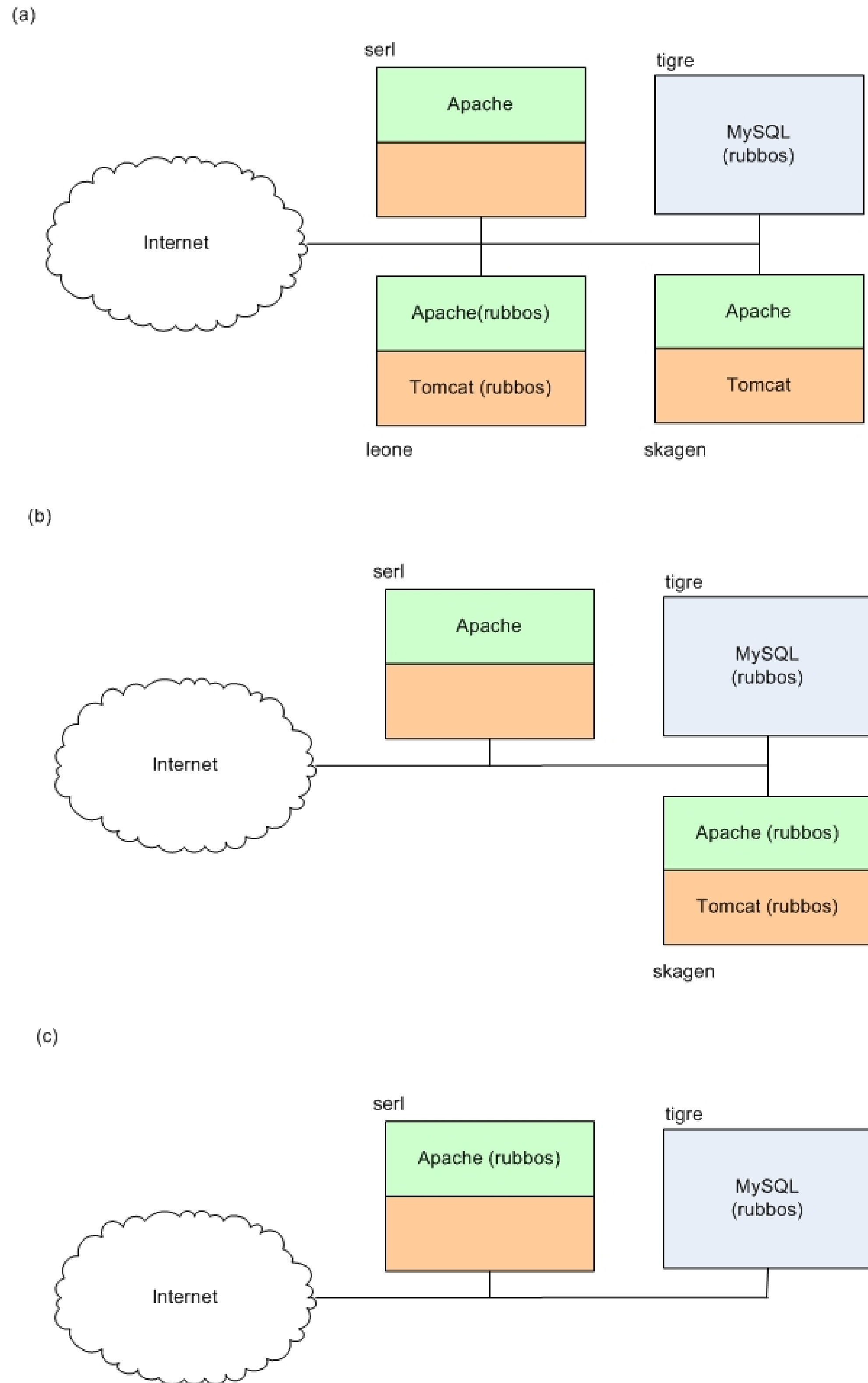


Figure 8.7: System State After two Consistent Failures

capabilities.

The architecture of system that we used in this experiment is similar to the architecture we used in the basic experiments. However, we only deployed one application on this system as shown in figure 8.7. The Rubbos application is deployed on machine Leone. The components on the other machines Serl and Skagen do not have any applications installed on their respective components. However, these components are still part of the system and can be used if required. The original state of the system before any failure is shown in figure 8.7(a).

We induce the failure of machine Leone in the system. Therefore, application Rubbos which is wholly deployed on Leone suffer a total loss of functionality. The similar recovery planning as we saw in basci experiments is performed. The relative times that the system took for recovery are given below.

Table 8.10: Recovery Process After First Failure

Time for each phase (in seconds)	
Failure Notification	0.1
Updating Models/Analysis of System	0.25
Developing Initial and Goal State	0.9
Recovery Planning	2.62
Plan Interpretation & Dispatch	0.2
Recovery Notification	0.1
Recovery Plan Execution	2.49
Recovery Results	0.1
Total Recovery Time	6.76

This recovery process restores the application Rubbos on Skagen, using its Apache and Tomcat instances. Now the application Rubbos is working on Skagen. The system state at this time is shown in figure 8.7(b).

When it is ensured that application Rubbos is working on Skagen, we induce the failure of machine Skagen in the system. The recover process started and following measurements are taken

Table 8.11: Recovery Process After Second Failure

Time for each phase (in seconds)	
Failure Notification	0.1
Updating Models/Analysis of System	0.24
Developing Initial and Goal State	0.9
Recovery Planning	5

The planner is given maximum of five seconds of cpu time to find the plan. However, as one can see there is no Tomcat available on Serl and Tigre. Therefore, the planner after using its five seconds is still not able to find a plan. Because of the unavailability of Tomcat no matter how much time we give to the planner, it still is not able to find a plan to restore Rubbos to its original configuration.

When the monitor detects that there is no plan given as output by the planner it invokes *Target Configuration Manager* (TCM) because the application Rubbos could not be restored to its original configuration. The TCM finds a new less strict configuration using the ACM, that only requires Apache and Mysql for the application Rubbos. The monitor starts again from model updating and the following measurements are performed.

Table 8.12: Recovery Process After Second Failure

Time for each phase (in seconds)	
Developing Initial and Goal State	0.3
Recovery Planning	2.68
Plan Interpretation & Dispatch	0.4
Recovery Notification	0.1
Recovery Plan Execution	0.19
Recovery Results	0.1
Total Recovery Time	10.13

The models are updated again. The monitor also writes a new problem file with the new configuration as the goal state. It then gives the planner the new problem file. At this time, the planner is able to find a plan for the new configuration and it

finds machine Serl to have all the dependencies for the new configuration. Therefore, the less strict configuration of application Rubbos is deployed on machine Serl and it starts operating with a reduced functionality. The total time to recover the application Rubbos is 10.13 seconds. The new state of the system is given in figure 8.7(c).

In brief, our failure recovery system 'Recover' is able handle multiple failures in the system, Moreover, it is also capable of recovering an application to a lesser strict configuration if for some reason the original configuration is not attainable.

8.4 Conclusion

This experimentation exercise led us to believe that automating a failure recovery process in component-based distributed systems is indeed a possibility. However, to achieve this the most important aspect is to model these systems with failure recovery in mind. We have shown with these experiments that how an off-the-self system can be modeled to achieve automated failure recovery. These experiments have also shown that to ease the job of an administrator we need to model systems at the micro-level i.e. configuration files because the complexity of these systems is really hidden in them. Modeling at the micro-level also provided the ability to specify high-level goals.

Our experiments have also shown that AI planning is indeed an effective tool for modeling and recovering a system from failure recovery. AI planning gave us several advantages over normal search based planning. First, it is fast as compared to the other search based techniques. Second, we are able to specify high-level goals and the planner itself found the low-level actions to achieve that goal. This is not possible in normal search techniques. Finally, planning give us a step by step plan optimizing the time and resource usage. It is very difficult and in come cases not possible for search techniques to do such optimizations.

Furthermore, our experiments have also shown that developing a failure recovery script at run-time is also possible. This run-time development of scripts also gave

several advantages. First, it is not required by an administrator to develop recovery scripts manually. Second, the development of script is very fast and is usually a fraction of a second when a plan is already in place. Third, the script is ordered in terms of the dependencies of the system so no action can take place unless a dependency of that action is already in place.

Finally, we stress that the results of these experiments are possible due to better modeling of the system for automated failure recovery. Better modeling always requires more time and cost e.g. the time to develop an AI planning domain. However, the cost and time spent on this modeling is amortized over time from the availability of the system.

Chapter 9

Future Research and Conclusion

9.1 Future Research

The research presented in this dissertation raised some interesting new problems. Some of the problems are directly stemmed from this research and can solve other problems in failure recovery. However, others are general that can help anybody doing research in component-based distributed systems.

9.1.1 Dealing with Failures during Failure Recovery

An important aspect of recovering from failures is to handle further failures in the system [10]. We have dealt with certain types of extended failures in this dissertation. However, the range of further failures is too broad. Therefore, the goal of this future research is to handle failures during failure recovery. As a recovering system is already in an inconsistent state, more failures can create a much more complicated situation. Therefore, these extended failures must be handled according to the extent of the failure and the state of the recovering system. In this research, we will find techniques such that the failure recovery in the systems is reliable. This research will find stable points in sequence of failures or extended failures when a recovery can be initiated. Moreover, it should deal with extended failures in the system and furthermore, it recovers the system without impacting the other unrelated components of the system.

9.1.2 Addition of More Resources in the System

Although we have assumeed that a machine and other resources cannot be recovered from a failure. However, in real world situations this is seldom true. Therefore, an interesting direction for future research is to take into account resources addition in future and to redeploy the applications. Even in normal scenarios this research is useful when more resources are added in the system and the applications needs to be redeployed to balance the load.

9.1.3 Developing a Framework for Developing Scripts for Reconfiguration

The development of scripts that can play a role of interface to the outside world is not very mature. Another interesting research topic is to develop a framework for developing these scripts so that one can easilt develop a new script without much problem. Presently one needs to be really proficient in the scripting langauge to develop good scripts.

9.1.4 Automated Mechanism to Develop a Planning Domain

Development of a planning domain is probably the most important in modeling a system for failure recovery. Therefore, good guidelines and tools must be developed to make this process easier for developer or administrator. This tood can be made such that the users dpecify the requirements and dependencies and the tood automatically generate a skeleton domain.

9.1.5 Dynamic Reconfiguration to Improve System Performance

Dynamic reconfiguration can be used in large scale distributed systems to improve system performance. There are some steps that are required to improve the system performance at runtime.

First, runtime performance improvement requires runtime performance measurement. Most of the techniques of measuring performance require the system to be taken off-line. There are no good techniques available for runtime performance measurement in live distributed systems. Therefore, the first goal of this project is to be develop tools and metrics to measure the performance of a given working system at any given time or between two time instances.

The second step is to find out the bottlenecks that cause the performance degradation. The initial approach to find the bottleneck is based on two kinds of probes: internal and external. External probe calculate the response time from the system and detects any performance degradation. Based on the results of the external probe the internal probe look at the components of the system and detects the component creating the performance degradation. Depending on the results from both the probes the third step is to perform the reconfiguration on the system to improve the system performance.

9.1.6 Distributed Systems Simulator

Development of component-based distributed systems is difficult because in most research environments it is difficult to evaluate their behavior in large scale settings. Therefore, most of the papers in component-based systems lack key performance data.

Although many simulators for distributed systems are available, however, most of these simulators simulate the levels below the application layer i.e. network and operating system levels. Therefore, in this future research we will develop a distributed systems simulator based on an extendible framework. This distributed simulator will be capable of having platform and language plugins to simulate the behavior on a certain platform or language. Therefore, the behavior of components developed in different languages for a variety of platforms can be tested and evaluated using this simulator. Initially, the goal is to develop a simulator to support homogeneous systems such as java-based systems and later we will extend it to heterogeneous systems.

9.1.7 Failure Forensics

It is also important for an administrator of the system to know about the failures and recovery performed by an automated system. Therefore, a way to store information about all the actions done by automated failure recovery system is needed. By having this kind of system the administrator can also diagnose problems that are not automatically solved by the recovery system itself.

9.2 Conclusion

In this dissertation we have presented an automated failure technique based on continuous monitoring and AI planning. Our technique is unique in a sense that it requires very little from the user during the running of the system. Therefore, if this technique is properly applied to the system after careful modeling, the systems will be able to automatically recover from failures without much human intervention. Having said we are not replacing the role of a human being in system administration. We are not at a point where we claim that our work is a panacea for all failure recovery in all types of distributed systems. Our technique can be applied to certain types of systems under certain failure scenario.

One of the most unique aspects of our technique is using AI planning for failure recovery. According to our literature survey nobody has used AI planning in failure recovery scenarios. However, AI planning has its own limitations in terms of representation of the system being modeled. Therefore, one cannot solely rely on AI planning. But AI planning, along with other techniques can be a powerful tool for failure recovery and anywhere where dynamic configuration of a system is required.

Bibliography

[1] Russell J. Abbott. Resourceful systems for fault tolerance, reliability, and safety. ACM Comput. Surv., 22(1):35–68, 1990.

[2] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for change: a reconfiguration language for distributed systems, 1994.

[3] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In Proceedings of the 17th IEEE Conference on Automated Software Engineering, pages 23–27, 2002.

[4] Sascha Alda, Markus Won, and Armin B. Cremers. Managing dependencies in component-based distributed applications. In Revised Papers from the International Workshop on Scientific Engineering for Distributed Java Applications, pages 143–154. Springer-Verlag, 2003.

[5] J. F. Allen and J. A. Koomen. Planning using a temporal world model. In J. Allen, J. Hendler, and A. Tate, editors, Readings in Planning, pages 559–565. Kaufmann, San Mateo, CA, 1990.

[6] James F. Allen. Planning as temporal reasoning. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, KR'91: Principles of Knowledge Representation and Reasoning, pages 3–14. Morgan Kaufmann, San Mateo, California, 1991.

[7] J.P.A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for corba. In Proceedings of the 3rd International Symposium on Distributed Objects and Applications, pages 197–207. IEEE Computer Society, 2001.

[8] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. In Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence, pages 39–46. IEEE Press, November 2003.

[9] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. A planning based approach to failure recovery in distributed systems. In Proceedings of the ACM SIGSOFT International Workshop on Self-Managed Systems (WOSS'04). ACM Press, Oct./Nov. 2004.

[10] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Dealing with failures during failure recovery of distributed systems. In Proceedings of the DEAS 2005, the ICSE 2005 Workshop on the Design and Evolution of Autonomic Application Software. ACM Press, May 2005.

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

[12] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. Technical Report 739, University of Newcastle upon Tyne, School of Computing Science, 2001.

[13] Thais Vasconcelos Batista and Noemi de La Rocque Rodriguez. Dynamic reconfiguration of component-based applications. In PDSE, pages 32–39, 2000.

[14] Jurgen Berghoff, Oswald Drobnik, Anselm Lingnau, and Christian Monch. Agent-based configuration management of distributed applications. In Proceedings of Third International Conference on Configurable Distributed Systems, 1996, pages 52–59. IEEE Computer Society Press, 1996.

[15] Mark Allen Boyd. Dynamic fault tree models: techniques for analysis of advanced fault tolerant computer systems. PhD thesis, Durham, NC, USA, 1992.

[16] Eric A. Brewer. Lessons from giant-scale services. IEEE Internet Computing, 5(4):46–55, 2001.

[17] A. Brown and D. Patterson. To err is human, 2001.

[18] A. Brown and D. A. Patterson. Embracing failure: A case for recovery-oriented computing (roc). In Proceedings of the High Performance Transaction Processing Symposium, October 2001.

[19] Tom Bylander. The computational complexity of propositional strips planning. Artif. Intell., 69(1-2):165–204, 1994.

[20] George Candea, Aaron B. Brown, Armando Fox, and David Patterson. Recovery-oriented computing: Building multitier dependability. Computer, 37(11):60–67, 2004.

[21] George Candea and Armando Fox. Crash-only software. In Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, May 18-21, 2003, Lihue (Kauai), Hawaii, USA, pages 67–72, 2003.

[22] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems, 19(3):332–383, August 2001.

[23] M. Castaldi, A. Carzaniga, P. Inverardi, and A.L. Wolf. A lightweight infrastructure for reconfiguring applications. In In B. Westfechtel, A. van der Hoek (Eds.): SCM 2001/2003, LNCS 2649, pages 231–244. LNCS, 2003.

[24] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings, pages 242–261, 2003.

[25] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43(2):225–267, 1996.

[26] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002), June 2002.

[27] Xuejun Chen and Martin Simons. A component framework for dynamic reconfiguration of distributed systems. In Proceedings of the IFIP/ACM Working Conference on Component Deployment, pages 82–96. Springer-Verlag, 2002.

[28] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, João Pedro Sousa, Bridget Spitnagel, and Peter Steenkiste. Using architectural style as a basis for system self-repair. In WICAS3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture, pages 45–59, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[29] S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. Aspen - automated planning and scheduling for space mission operations.

[30] Jonathan E. Cook and Jeffrey A. Dage. Highly reliable upgrading of components. In Proceedings of the 21st international conference on Software engineering, pages 203–212. IEEE Computer Society Press, 1999.

[31] Lisa Cox and Harry S. Delugach. Dependency analysis using conceptual graphs. In Proceedings of the 9th International Conference on Conceptual Structures, ICCS 2001, Stanford, CA, USA, July 30-August 3, 2001, volume 2120 of Lecture Notes in Computer Science. Springer, 2001.

[32] Flavin Cristian. Understanding fault-tolerant distributed systems. Commun. ACM, 34(2):56–78, 1991.

[33] Yi Cui and Klara Nahrstedt. Qos-aware dependency management for component-based systems. In HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01), page 127, Washington, DC, USA, 2001. IEEE Computer Society.

[34] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In WOSS '02: Proceedings of the first workshop on Self-healing systems, pages 21–26, New York, NY, USA, 2002. ACM Press.

[35] Xavier Défago, Naohiro Hayashibara, and Takuya Katayama. On the design of a failure detection service for large scale distributed systems. In Proc. Int'l Symp. Towards Peta-Bit Ultra-Networks (PBit 2003), pages 88–95, Ishikawa, Japan, September 2003.

[36] Yixin Diao, Joseph L. Hellerstein, Sujay S. Parekh, Rean Griffith, Gail E. Kaiser, and Dan B. Phung. Self-managing systems: A control theory foundation. In 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), 4-7 April 2005, Greenbelt, MD, USA, pages 441–448, 2005.

[37] Donn DiNunno. Quantifying performance loss. it performance engineering & measurement strategies. Meta Group, October 2000.

[38] J. Dunagan, N. Harvey, M. Jones, D. Kostic, M. Theimer, and A. Wolman. Fuse: Lightweight guaranteed distributed failure notification, 2004.

[39] Christian Ensel. A scalable approach to automated service dependency modeling in heterogeneous environments. In 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), 4-7 September 2001, Seattle, WA, USA, Proceedings, pages 128–139, 2001.

[40] Peter Feiler and Jun Li. Consistency in dynamic reconfiguration. In Proceedings of the Fourth International Conference on Configurable Distributed Systems, pages 189–196, 1998.

[41] Pascal Felber, Xavier Dfago, Rachid Guerraoui, and Philipp Oser. Failure detectors as first class objects. In DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications, page 132, Washington, DC, USA, 1999. IEEE Computer Society.

[42] Manual For. Using the sipe-2 planning system.

[43] David Garlan, Vahe Poladian, Bradley Schmerl, and João Pedro Sousa. Task-based self-adaptation. In WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, pages 54–57, New York, NY, USA, 2004. ACM Press.

[44] David Garlan, Bradley Schmerl, and Jichuan Chang. Using gauges for architecture-based monitoring and adaptation. In Working Conference on Complex and Dynamic Systems Architecture, December 2001. In press.

[45] John C. Georgas, André van der Hoek, and Richard N. Taylor. Architectural runtime configuration management in support of dependable self-adaptive software. In WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems, pages 1–6, New York, NY, USA, 2005. ACM Press.

[46] Selvin George, David Evans, and Lance Davidson. A biologically inspired programming model for self-healing systems. In WOSS '02: Proceedings of the first workshop on Self-healing systems, pages 102–104. ACM Press, 2002.

[47] A. Gerevini and I. Serina. Lpg: a planner based on planning graphs with action costs. In Proceedings of the Sixth Int. Conference on AI Planning and Scheduling (AIPS'02), pages 12–22. AAAI Press, 2002.

[48] Jim Gray. Why do computers stop and what can be done about it? In Symposium on Reliability in Distributed Software and Database Systems, pages 3–12, 1986.

[49] Philip N. Gross, Suhit Gupta, Gail E. Kaiser, Gaurav S. Kc, and Janak J. Parekh. An active events model for systems monitoring. In Working Conference on Complex and Dynamic Systems Architecture, December 2001. In press.

[50] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the software dock. In International Conference on Software Engineering, pages 174–183, 1999.

[51] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In Proceedings of Third International Conference on Configurable Distributed Systems, 1996, pages 70–80. IEEE Computer Society Press, 1996.

[52] N. Hayashibara, X. Defago, and T. Katayama. Two-ways adaptive failure detection with the $\varphi-$failure detector, 2003.

[53] Malte Helmert. Complexity results for standard benchmark domains in planning. Artif. Intell., 143(2):219–262, 2003.

[54] Gail Kaiser, Phil Gross, Gaurav Kc, Janak Parekh, and Giuseppe Valetto. An approach to autonomizing legacy systems, in workshop on self-healing, adaptive and self-managed systems. In Workshop on Self-Healing, Adaptive and Self-MANaged Systems, June 2002.

[55] A. Keller and G. Kar. Dynamic dependencies in application service management, 2000.

[56] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. Computer, 36(1):41–50, 2003.

[57] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. In IPDPS, 2003.

[58] T. Kichkaylo, A. Ivan, and V. Karamcheti. Sekitei: An ai planner for constrained component deployment in wide-area networks. Technical Report NYU-CS-TR851, New York University, 2004.

[59] Reinhard Klemm and Navjot Singh. Automatic failure detection and recovery for java servers. In Maintenance and Reliability Conference (MARCON), Gatlinburg, Tennessee, May 2001.

[60] J. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, and P. Devanbu. The willow survivability architecture, 2001.

[61] Fabio Kon and Roy H. Campbell. Dependence Management in Component-Based Distributed Systems. IEEE Concurrency, 8(1):26–36, January-March 2000.

[62] Philip Koopman. Elements of the self-healing system problem space. Workshop on Architecting Dependable Systems/WADS03, May 2003.

[63] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering, 16(11):1293–1306, nov 1990.

[64] Jean-Claude Laprie. How much is safety worth? In IFIP Congress (3), pages 251–253, 1994.

[65] Magnus Larsson and Ivica Crnkovic. Component configuration management. In ECOOP Conference, Workshop on Component Oriented Programming, Nice, France, January 2000.

[66] Jun Li. Monitoring of component-based systems. Technical Report HPL-2002-25R1-20030606, Hewlett-Packard Development Company, L.P., June 2003.

[67] M. Little and S. Wheater. Building configurable applications in java, 1998.

[68] Derek Long and Maria Fox. Pddl2.1: An extension to pddl for expressing temporal planning domains. Journal of Artificial Intelligence Research (JAIR), Special Issue on the 3rd International Planning Competition, 20(1):61–124, 2003.

[69] M.Endler and J.Wei. Programming generic dynamic reconfigurations for distributed applications. In Proceedings of the International Workshop Configurable Distributed Systems, pages 68–79. IEE, 1992.

[70] K. Mills, S. Rose, S. Quirolgico, M. Britton, and C. Tan. An autonomic failure-detection algorithm. SIGSOFT Softw. Eng. Notes, 29(1):79–83, 2004.

[71] Nicola Muscettola. Hsts: Integrating planning and scheduling. Technical Report CMU-RI-TR-93-05, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, March 1993.

[72] Netcraft.com. October 2005 web server survey, 2005.

[73] Allen Newell, J. C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In IFIP Congress, pages 256–264, 1959.

[74] David L. Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In USENIX Symposium on Internet Technologies and Systems, 2003.

[75] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software, 1999.

[76] N. De Palma and B. Riveill. Dynamic reconfiguration of agent-based applications, 1999.

[77] George A. Papadopuolos and Farhad Arbab. Dynamic reconfiguration in coordination languages. In HPCN Europe, pages 197–206, 2000.

[78] Joon Park and Pratheep Chandramohan. Static vs. dynamic recovery models for survivable distributed systems. In HICSS, 2004.

[79] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, and Noah Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. In UC Berkeley Computer Science Technical Report UCB/CSD-02-1175, Berkeley, CA, March 2002. U.C. Berkeley.

[80] J. Paulo, A. Almeida, M. Wegdam, L. Ferreira Pires, and M. Sinderen. An approach to dynamic reconfiguration of distributed systems based on object-middleware. In Proceedings of the 19th Brazilian Symposium on Computer Networks (SBRC 2001), 2001.

[81] J. Scott Penberthy and Daniel S. Weld. Temporal planning with continuous change. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), volume 2, pages 1010–1015, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[82] Planet: European network of excellence in ai planning.

[83] B. Randell. System structure for software fault tolerance. In Proceedings of the international conference on Reliable software, pages 437–449, 1975.

[84] D. Reilly, A. Taleb-Bendiab, A. Laws, and N. Badr. An instrumentation and control-based approach for distributed application management and adaptation. In WOSS '02: Proceedings of the first workshop on Self-healing systems, pages 61–66, New York, NY, USA, 2002. ACM Press.

[85] Christopher Roblee, Vincent Berk, and George Cybenko. Large-scale autonomic server monitoring using process query systems. In Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC 2005), Seatle, WA, 2005.

[86] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach (second edition). Prentice-Hall, Englewood Cliffs, NJ, 2003.

[87] M. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A. Wolf. Reconfiguration in the enterprise javabean component model. In Component Deployment, volume 2370 of Lecture Notes in Computer Science. Springer, 2002.

[88] R. K. Scott, J. W. Gault, and D. F. McAllister. Fault-tolerant software reliability modeling. IEEE Trans. Softw. Eng., 13(5):582–592, 1987.

[89] N. Serrano, F. Alonso, J. M. Sarriegi, J. Santos, and I. Ciordia. A new undo function for web-based management information systems. IEEE Internet Computing, 9(2):38–44, 2005.

[90] Michael E. Shin and Daniel Cooke. Connector-based self-healing mechanism for components of a reliable system. In DEAS '05: Proceedings of the 2005 workshop

on Design and evolution of autonomic application software, pages 1–7, New York, NY, USA, 2005. ACM Press.

[91] S. Shrivastava and S. Wheater. Architectural support for dynamic reconfiguration of large scale distributed applications, 1998.

[92] C. Soules, J. Appavoo, K. Hui, D. Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration, 2003.

[93] J. Stafford, D. Richardson, and A. Wolf. Chaining: A software architecture dependence analysis technique, 1997.

[94] Paul Stelling, Cheryl DeMatteis, Ian T. Foster, Carl Kesselman, Craig A. Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. Cluster Computing, 2(2):117–128, 1999.

[95] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, and L. Iftode. Recovering internet service sessions from operating system failures. IEEE Internet Computing, 9(2):17–27, 2005.

[96] Tamara Sumner, Sonal Bhushan, Faisal Ahmad, and Qianyi Gu. Designing a language for creating conceptual browsing interfaces for digital libraries. In JCDL '03: Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital libraries, pages 258–260, Washington, DC, USA, 2003. IEEE Computer Society.

[97] Matthias Tichy, Holger Giese, Daniela Schilling, and Wladimir Pauls. Computing optimal self-repair actions: damage minimization versus repair time. In WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems, pages 7–6, New York, NY, USA, 2005. ACM Press.

[98] Giuseppe Valetto, Gail E. Kaiser, and Gaurav S. Kc. A mobile agent approach to process-based dynamic adaptation of complex software systems. In Proceedings of the 8th European Workshop on Software Process Technology, pages 102–116. Springer-Verlag, 2001.

[99] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. Journal of Experimental and Theoretical Artificial Intelligence, 7(1):81–120, 1995.

[100] S. A. Vere. Planning in time: Windows and durations for activities and goals. In J. Allen, J. Hendler, and A. Tate, editors, Readings in Planning, pages 297–318. Kaufmann, San Mateo, CA, 1990.

[101] Don Welch. Building self-reconfiguring distributed systems using compensating reconfiguration. In Proceedings of the 4th International Conference on Configurable Distributed Systems, pages 18–25. IEEE Computer Society Press, 1998.

[102] Michel Wermelinger. Towards a chemical model for software architecture reconfiguration. In Proceedings of the 4th International Conference on Configurable Distributed Systems, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1998. IEEE.

[103] Michel Wermelinger and Jose Luiz Fiadeiro. Algebraic software architecture reconfiguration. In ESEC / SIGSOFT FSE, pages 393–409, 1999.

[104] Michel Wermelinger, Antonia Lopes, and Jose Luiz Fiadeiro. A graph based architectural (re)configuration language. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pages 21–32. ACM Press, 2001.

[105] K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer. A system model for dynamically reconfigurable software. IBM Systems Journal, 42(1):45–59, 2003.

[106] Torres Wilfredo. Software fault tolerance: A tutorial. Technical report, 2000.

[107] Jie Xu and Brian Randell. The t/(n-1)-vp approach to fault-tolerant software.

Appendix A

Initial and Goal State

```
(
define ( problem testp1 )
( :domain final_domain )
(:objects
leone serl serl-back skagen tigre - machine
httpd - file
libphp4 jsp mime alias rpc cache benchmark security ssl setenvif imap
dr - module
libphp4_service mime_service alias_service rpc_service cache_service
benchmark_service security_service imap_service jsp_service
ssl_service setenvif_service - service
mod_jk mod_jk2 mod_proxy - connector
apache_skagen apache_serl-back apache_leone apache_serl - apache
tomcat_skagen tomcat_leone - tomcat
rubbos sms webcal - application
)
(:init
(= (max-machine-load leone) 100)
(= (max-machine-load serl) 100)
(= (max-machine-load serl-back) 100)
(= (max-machine-load skagen) 100)
(= (max-machine-load tigre) 100)
(= (current-machine-load leone) 10)
(= (current-machine-load serl) 10)
(= (current-machine-load serl-back) 10)
(= (current-machine-load skagen) 10)
(= (current-machine-load tigre) 10)
(= (time-to-start-apache apache_skagen) 2000)
(= (time-to-start-tomcat tomcat_skagen) 3000)
(= (time-to-start-apache apache_serl-back) 1000)
(= (time-to-start-apache apache_leone) 9000)
(= (time-to-start-tomcat tomcat_leone) 5000)
(= (time-to-start-apache apache_serl) 7000)
```

```
(= (application-count apache_skagen) 1)
(= (application-count apache_serl-back) 1)
(= (application-count apache_leone) 0)
(= (application-count apache_serl) 1)
(= (time-to-restart-apache apache_skagen) 3000)
(= (time-to-restart-tomcat tomcat_skagen) 4000)
(= (time-to-restart-apache apache_serl-back) 1200)
(= (time-to-restart-apache apache_leone) 11000)
(= (time-to-restart-tomcat tomcat_leone) 2100)
(= (time-to-restart-apache apache_serl) 9000)
(= (apache-load apache_skagen) 50)
(= (tomcat-load tomcat_skagen) 20)
(= (apache-load apache_serl-back) 10)
(= (apache-load apache_leone) 10)
(= (tomcat-load tomcat_leone) 20)
(= (apache-load apache_serl) 50)
(= (time-to-import-application-in-apache rubbos) 1)
(= (time-to-import-application-in-tomcat rubbos) 1)
(= (time-to-import-application-in-apache sms) 1)
(= (time-to-import-application-in-tomcat sms) 1)
(= (time-to-import-application-in-apache webcal) 1)
(= (time-to-import-application-in-tomcat webcal) 1)

    (machine-working leone)
(application-available sms leone)
(application-available rubbos leone)
(application-available webcal leone)
(apache-installation-available apache_leone leone)
(tomcat-installation-available tomcat_leone leone)

    (machine-failed serl)
(application-available sms serl)
(application-available rubbos serl)
(application-available webcal serl)
(apache-installation-available apache_serl serl)

    (machine-working serl-back)
(application-available sms serl-back)
(application-available rubbos serl-back)
(application-available webcal serl-back)
(apache-installation-available apache_serl-back serl-back)

    (machine-working skagen)
(application-available sms skagen)
(application-available rubbos skagen)
(application-available webcal skagen)
(apache-installation-available apache_skagen skagen)
```

(tomcat-installation-available tomcat_skagen skagen)

(machine-working tigre)
(application-available sms tigre)
(application-available rubbos tigre)
(application-available webcal tigre)

(apache-working apache_skagen skagen)
(apache-has-configuration-file apache_skagen httpd)
(apache-has-module-installation apache_skagen setenvif skagen)
(apache-module-provides-service apache_skagen setenvif setenvif_service)
(apache-has-installed-module apache_skagen ssl skagen)
(apache-providing-service apache_skagen skagen ssl_service)
(apache-has-installed-module apache_skagen libphp4 skagen)
(apache-providing-service apache_skagen skagen libphp4_service)
(connector-available-in-apache mod_jk apache_skagen skagen)
(connector-available-in-apache mod_proxy apache_skagen skagen)

(tomcat-working tomcat_skagen skagen)
(connector-available-in-tomcat mod_jk tomcat_skagen skagen)
(connector-available-in-tomcat mod_proxy tomcat_skagen skagen)

(apache-working apache_serl-back serl-back)
(apache-has-configuration-file apache_serl-back httpd)
(apache-has-module-installation apache_serl-back ssl serl-back)
(apache-module-provides-service apache_serl-back ssl ssl_service)
(apache-has-installed-module apache_serl-back jsp serl-back)
(apache-providing-service apache_serl-back serl-back jsp_service)
(apache-has-installed-module apache_serl-back libphp4 serl-back)
(apache-providing-service apache_serl-back serl-back libphp4_service)
(connector-available-in-apache mod_jk apache_serl-back serl-back)
(connector-available-in-apache mod_proxy apache_serl-back serl-back)

(apache-working apache_leone leone)
(apache-has-configuration-file apache_leone httpd)
(apache-has-module-installation apache_leone jsp leone)
(apache-module-provides-service apache_leone jsp jsp_service)
(apache-has-installed-module apache_leone ssl leone)
(apache-providing-service apache_leone leone ssl_service)
(connector-available-in-apache mod_jk apache_leone leone)
(connector-available-in-apache mod_proxy apache_leone leone)
(connector-available-in-apache mod_jk2 apache_leone leone)

(tomcat-working tomcat_leone leone)
(connector-available-in-tomcat mod_jk2 tomcat_leone leone)
(connector-available-in-tomcat mod_proxy tomcat_leone leone)

```
(apache-has-configuration-file apache_serl httpd)
(apache-has-module-installation apache_serl setenvif serl)
(apache-module-provides-service apache_serl setenvif setenvif_service)
(apache-has-installed-module apache_serl ssl serl)
(apache-providing-service apache_serl serl ssl_service)
(apache-has-installed-module apache_serl libphp4 serl)
(apache-providing-service apache_serl serl libphp4_service)
(connector-available-in-apache mod_jk apache_serl serl)
(connector-available-in-apache mod_proxy apache_serl serl)

(application-ready-in-tomcat rubbos tomcat_leone leone)
(tomcat-configured tomcat_leone leone)

(application-ready-in-apache sms apache_skagen skagen)
(apache-configured apache_skagen skagen)
(application-ready-in-tomcat sms tomcat_skagen skagen)
(tomcat-configured tomcat_skagen skagen)

(application-ready-in-apache webcal apache_serl-back serl-back)
(apache-configured apache_serl-back serl-back)
(application-requires-service rubbos ssl_service)
(application-requires-service webcal libphp4_service)
)
(:goal
(and
(application-ready-1 rubbos)
)
)
(:metric minimize (time-to-start-apache))
)
```

Appendix B

Plan

; Version LPG-td-1.0
; Seed 15958803
; Command line: lpg-td-1.0 -o final_domain.pddl -f testp1.pddl -n 5 -cputime 20
; Problem testp1.pddl
; Time 3.62
; Search time 0.00
; Parsing time 3.60
; Mutex time 0.01
; Quality 0.60

Time 3.62

0.0003: (RECONFIGURE-APACHE SKAGEN APACHE_SKAGEN RUBBOS HTTPD)
1.0005: (IMPORT-APPLICATION-IN-APACHE SKAGEN APACHE_SKAGEN RUBBOS)
2.0008: (UPDATE-CONFIGURATION-FILE-FOR-RECONFIGURATION-ADDVIRTUALHOST SKAGEN APACHE_SKAGEN HTTPD RUBBOS SMS)
7.0010: (RESTART-APACHE-WITH-VIRTUALHOST SKAGEN APACHE_SKAGEN RUBBOS SMS)
3007.0012: (ADD-CONNECTIVITY SKAGEN LEONE APACHE_SKAGEN TOMCAT_LEONE MOD_PROXY RUBBOS)
3008.0015: (START-SYSTEM-1 SKAGEN LEONE RUBBOS APACHE_SKAGEN TOMCAT_LEONE MOD_PROXY SSL_SERVICE LIBPHP4)

Appendix C

Failure Recovery Planning Domain

```
(define (domain final_domain)
(:requirements :strips :typing :equality :adl :fluents)
  (:types
machine apache tomcat file application module connector
service mysql software - object
)

(:predicates
(application-available ?app - application ?ma - machine)
(application-requires-service ?app - application ?s - service)
(application-ready-5 ?app - application ?ap - apache ?s - service
?t - tomcat ?con - connector)
(application-ready-4 ?app - application ?ap - apache ?t - tomcat
?con - connector)
(application-ready-3 ?app - application ?ap - apache ?t - tomcat)
(application-ready-2a ?app - application ?ap - apache)
(application-ready-2a-with-service ?app - application ?ap - apache
?s - service)
(application-ready-2b ?app - application ?t - tomcat)
(application-ready-1 ?app - application)
(application-ready-3-with-connectivity ?app - application
?ap - apache ?t - tomcat)
(application-ready-3-with-service ?app - application
?ap - apache ?t - tomcat ?s - service)
(application-ready-3-with-connectivity-and-service
?app - application ?ap - apache ?t - tomcat ?s - service)
(application-ready-1a ?app - application)
(application-ready-1b ?app - application)
(application-ready-in-apache ?app - application ?ap - apache
?ma - machine)
(application-ready-in-tomcat ?app - application ?t - tomcat
```

```
?ma - machine)
(machine-working ?ma - machine)
(machine-failed ?ma - machine)
(virtualhostadded-in-apache ?ap - apache ?app1 - application
?app2 - application)
(import-application-to-apache ?app - application ?ap - apache
?ma - machine)
(application-available-in-apache ?app - application
?ap - apache ?ma - machine)
(apache-working ?ap - apache ?ma - machine)
(apache-has-configuration-file ?ap - apache ?f - file)
(apache-has-installed-module ?ap - apache ?mo - module ?ma - machine)
(apache-has-module-installation ?ap - apache ?mo - module ?ma - machine)
(apache-module-provides-service ?ap - apache ?mo - module ?s - service)
(apache-providing-service ?ap - apache ?ma - machine ?s - service)
(apache-configured ?ap - apache ?ma - machine)
(apache-installation-available ?ap - apache ?ma - machine)
(apache-configuration-file-require-modification ?ap - apache
?ma - machine ?f - file)
(apache-configuration-file-updated ?ap - apache ?ma - machine ?f - file)
      (connectivity-available ?ap - apache ?ma1 - machine ?t - tomcat
?ma2 - machine ?con - connector)
(connector-configuration-required ?ap - apache ?ma - machine
?t - tomcat ?ma - machine ?con - connector)
(connection-configured ?ap - apache ?ma - machine ?t - tomcat
?ma - machine ?con - connector)
(apache-reconfigured ?ap - apache ?m - machine)
(virtualhostadded ?c - apache ?a1 - application ?a2 - application)
(apache-module-require-configuration ?ap - apache
?ma - machine ?mo - module)
(connector-available-in-apache ?con - connector ?ap - apache
?ma - machine)
(connector-available-in-tomcat ?con - connector ?t - tomcat
?ma - machine)
(application-available-in-tomcat ?app - application ?t - tomcat
?ma - machine)
(tomcat-installation-available ?t - tomcat ?ma - machine)
(import-application-to-tomcat ?app - application ?t - tomcat
?ma - machine )
(tomcat-configured ?t - tomcat ?ma - machine)
(tomcat-working ?t - tomcat ?ma - machine)
)

(:functions
(time-to-start-apache ?ap - apache)
(time-to-restart-apache ?ap - apache)
(time-to-start-tomcat ?t - tomcat)
```

```
(time-to-restart-tomcat ?t - tomcat)
(max-machine-load ?ma - machine)
(current-machine-load ?ma - machine)
(apache-load ?ap - apache)
(tomcat-load ?t - tomcat)
(time-to-import-application-in-apache ?app - application)
(time-to-import-application-in-tomcat ?app - application)
(application-count ?a - apache)
)


(:durative-action add-module-to-apache
:parameters (?ma - machine ?ap - apache ?mo - module ?s - service)
:duration
   (= ?duration 1)
:condition
(and
(at start (apache-installation-available ?ap ?ma))
(at start (apache-has-module-installation ?ap ?mo ?ma))
(at start (not (apache-has-installed-module ?ap ?mo ?ma )))
(at start (apache-module-provides-service ?ap ?mo ?s))
)
:effect
(and
(at end (apache-module-require-configuration ?ap ?ma ?mo))
)
)

(:durative-action configure-module-in-apache
:parameters (?ma - machine ?ap - apache ?mo - module ?s - service)
:duration
   (= ?duration 1)
:condition
(and
(at start (apache-module-require-configuration ?ap ?ma ?mo))
(at start (apache-module-provides-service ?ap ?mo ?s))
)
:effect
(and
(at end (apache-has-installed-module ?ap ?mo ?ma))
)
)

(:durative-action configure-apache
:parameters (?ma - machine ?ap - apache ?app - application ?f - file)
:duration
   (= ?duration 5)
```

```
:condition
(and
(at start (apache-installation-available ?ap ?ma))
(at start (not (apache-working ?ap ?ma)))
(at start (apache-has-configuration-file ?ap ?f))
)
:effect
(and
(at end (apache-configuration-file-require-modification ?ap ?ma ?f))
(at end (import-application-to-apache ?app ?ap ?ma))
)
)

(:durative-action import-application-in-apache
:parameters (?ma - machine ?ap - apache ?app - application)
:duration
(= ?duration (time-to-import-application-in-apache ?app))
:condition
(and
(at start (not (application-available-in-apache ?app ?ap ?ma )))
(at start (import-application-to-apache ?app ?ap ?ma))
        (at start (application-available ?app ?ma))
)
:effect
(and
(at end (application-available-in-apache ?app ?ap ?ma))


)
)
(:durative-action update-configuration-file-for-
configuration-settingup-application
:parameters (?ma - machine ?ap - apache ?f - file ?app - application)
:duration
   (= ?duration 1)
:condition
(and
(at start (apache-configuration-file-require-modification ?ap ?ma ?f))
(at start (application-available-in-apache ?app ?ap ?ma))
)
:effect
(and
(at end (apache-configuration-file-updated ?ap ?ma ?f))
(at end (apache-configured ?ap ?ma))

)
)
```

```
(:durative-action start-apache-with-service
:parameters (?ma - machine ?ap - apache ?app - application
?s - service ?mo - module)
:duration
   (= ?duration (time-to-start-apache ?ap))
:condition
(and
(at start (apache-configured ?ap ?ma))
(at start (not (apache-working ?ap ?ma)))
(at start (machine-working ?ma))
(at start (application-available-in-apache ?app ?ap ?ma))
(at start (> (- (max-machine-load ?ma) (current-machine-load ?ma))
(apache-load ?ap) ))
(at start (apache-has-installed-module ?ap ?mo ?ma))
(at start (apache-module-provides-service ?ap ?mo ?s ))
(at start (application-requires-service ?app ?s))
)
:effect
(and
(at end (apache-working ?ap ?ma))
      (at end (apache-providing-service ?ap ?ma ?s))
(at end (application-ready-in-apache ?app ?ap ?ma))


(at start (increase (current-machine-load ?ma) (apache-load ?ap)))
)
)
(:durative-action start-apache
:parameters (?ma - machine ?ap - apache ?app - application)
:duration
   (= ?duration (time-to-start-apache ?ap))
:condition
(and
(at start (apache-configured ?ap ?ma))
(at start (not (apache-working ?ap ?ma)))
(at start (machine-working ?ma))
(at start (application-available-in-apache ?app ?ap ?ma))
(at start (> (- (max-machine-load ?ma) (current-machine-load ?ma))
(apache-load ?ap) ))
)
:effect
(and
(at end (apache-working ?ap ?ma))
      (at end (application-ready-in-apache ?app ?ap ?ma))
(at start (increase (current-machine-load ?ma) (apache-load ?ap)))
)
)
```

```
(:durative-action reconfigure-apache
:parameters (?ma - machine ?ap - apache ?app - application ?f - file)
:duration
   (= ?duration 1)
:condition
(and
(at start (apache-installation-available ?ap ?ma))
(at start (apache-working ?ap ?ma))
(at start (apache-has-configuration-file ?ap ?f))
)
:effect
(and
(at end (apache-configuration-file-require-modification ?ap ?ma ?f))
(at end (import-application-to-apache ?app ?ap ?ma))
)
)
(:durative-action update-configuration-file-for-
reconfiguration-addvirtualhost
:parameters (?ma - machine ?ap - apache ?f - file
?app1 - application ?app2 - application)
:duration
   (= ?duration 5)
:condition
(and
(at start (apache-configuration-file-require-modification ?ap ?ma ?f))
(at start (application-available-in-apache ?app1 ?ap ?ma))
(at start (application-ready-in-apache ?app2 ?ap ?ma))
(at start (not (virtualhostadded-in-apache ?ap ?app1 ?app2)))
)
:effect
(and
(at end (virtualhostadded-in-apache ?ap ?app1 ?app2))
(at end (apache-configuration-file-updated ?ap ?ma ?f))
(at end (apache-configured ?ap ?ma))

)
)
(:durative-action restart-apache-with-service
:parameters (?ma - machine ?ap - apache ?app1 - application
?s - service ?mo - module)
:duration
   (= ?duration (time-to-restart-apache ?ap))
:condition
(and
;;(at start (virtualhostadded-in-apache ?ap ?app1 ?app2))
(at start (apache-working ?ap ?ma))
(at start (apache-has-installed-module ?ap ?mo ?ma))
```

```
(at start (apache-module-provides-service ?ap ?mo ?s ))
(at start (application-available-in-apache ?app1 ?ap ?ma))
(at start (machine-working ?ma))
(at start (> (- (max-machine-load ?ma) (current-machine-load ?ma))
(apache-load ?ap) ))
(at start (application-requires-service ?app1 ?s))
)
:effect
(and
(at end (apache-working ?ap ?ma))
(at end (apache-providing-service ?ap ?ma ?s))
(at end (application-ready-in-apache ?app1 ?ap ?ma))
;;(at end (application-ready-in-apache ?app2 ?ap ?ma))
(at start (increase (current-machine-load ?ma) (apache-load ?ap)))
)
)
(:durative-action restart-apache-with-virtualhost
:parameters (?ma - machine ?ap - apache ?app1 - application
?app2 - application)
:duration
   (= ?duration (time-to-restart-apache ?ap))
:condition
(and
(at start (virtualhostadded-in-apache ?ap ?app1 ?app2))
(at start (apache-working ?ap ?ma))
(at start (machine-working ?ma))
(at start (> (- (max-machine-load ?ma) (current-machine-load ?ma))
(apache-load ?ap) ))
)
:effect
(and
(at end (apache-working ?ap ?ma))
(at end (application-ready-in-apache ?app1 ?ap ?ma))
(at end (application-ready-in-apache ?app2 ?ap ?ma))
(at start (increase (current-machine-load ?ma) (apache-load ?ap)))
)
)
(:durative-action restart-apache
:parameters (?ma - machine ?ap - apache ?app1 - application)
:duration
   (= ?duration (time-to-restart-apache ?ap))
:condition
(and

(at start (apache-working ?ap ?ma))
(at start (machine-working ?ma))
(at start (> (- (max-machine-load ?ma) (current-machine-load ?ma))
```

```
(apache-load ?ap) ))
(at start (application-available-in-apache ?app1 ?ap ?ma))
(at start (< (application-count ?ap) 1))
)
:effect
(and
(at end (apache-working ?ap ?ma))
(at end (application-ready-in-apache ?app1 ?ap ?ma))
(at start (increase (application-count ?ap) 1))
(at start (increase (current-machine-load ?ma) (apache-load ?ap)))
)
)
(:durative-action configure-tomcat
:parameters (?ma - machine ?t - tomcat ?app - application )
:duration
   (= ?duration 1)
:condition
(and
(at start (tomcat-installation-available ?t ?ma))
(at start (not (tomcat-working ?t ?ma)))

)
:effect
(and
(at end (import-application-to-tomcat ?app ?t ?ma))
)
)
(:durative-action reconfigure-tomcat
:parameters (?ma - machine ?t - tomcat ?app - application )
:duration
   (= ?duration 5)
:condition
(and
(at start (tomcat-installation-available ?t ?ma))
(at start (tomcat-working ?t ?ma))

)
:effect
(and
(at end (import-application-to-tomcat ?app ?t ?ma))
)
)
(:durative-action import-application-in-tomcat
:parameters (?ma - machine ?t - tomcat ?app - application)
:duration
(= ?duration (time-to-import-application-in-tomcat ?app))
:condition
```

```
(and
(at start (application-available ?app ?ma))
(at start (import-application-to-tomcat ?app ?t ?ma))
(at start (not (application-available-in-tomcat ?app ?t ?ma)))
)
:effect
(and
(at end (application-available-in-tomcat ?app ?t ?ma))
(at end (tomcat-configured ?t ?ma))

)
)
(:durative-action start-tomcat
:parameters (?ma - machine ?t - tomcat ?app - application)
:duration
   (= ?duration (time-to-start-tomcat ?t))
:condition
(and
(at start (tomcat-configured ?t ?ma))
(at start (not (tomcat-working ?t ?ma)))
(at start (machine-working ?ma))
(at start (application-available-in-tomcat ?app ?t ?ma))
(at start (> (- (max-machine-load ?ma) (current-machine-load ?ma))
(tomcat-load ?t) ))

)
:effect
(and
(at end (tomcat-working ?t ?ma))
(at end (application-ready-in-tomcat ?app ?t ?ma))
(at start (increase (current-machine-load ?ma) (tomcat-load ?t)))
)
)
(:durative-action restart-tomcat
:parameters (?ma - machine ?t - tomcat ?app - application)
:duration
   (= ?duration (time-to-start-tomcat ?t))
:condition
(and
(at start (tomcat-configured ?t ?ma))
(at start (tomcat-working ?t ?ma))
(at start (machine-working ?ma))
(at start (application-available-in-tomcat ?app ?t ?ma))
(at start (> (- (max-machine-load ?ma) (current-machine-load ?ma))
(tomcat-load ?t) ))

)
```

```
:effect
(and
(at end (tomcat-working ?t ?ma))
(at end (application-ready-in-tomcat ?app ?t ?ma))
(at start (increase (current-machine-load ?ma) (tomcat-load ?t)))
)
)
(:durative-action add-connectivity
:parameters (?ma1 - machine ?ma2 - machine ?ap - apache
?t - tomcat ?con - connector ?app1 - application)
:duration
(= ?duration 1)
:condition
(and
(at start (tomcat-configured ?t ?ma2))
(at start (apache-configured ?ap ?ma1))
(at start (connector-available-in-apache ?con ?ap ?ma1 ))
(at start (connector-available-in-tomcat ?con ?t ?ma2 ))
(at start (application-ready-in-apache ?app1 ?ap ?ma1))
(at start (application-ready-in-tomcat ?app1 ?t ?ma2))

)
:effect
(and
        (at end (connectivity-available ?ap ?ma1 ?t ?ma2 ?con))
)
)
(:durative-action start-system-1
:parameters (?ma1 - machine ?ma2 - machine ?app - application
?ap - apache ?t - tomcat ?con - connector ?s - service
?mo -  module)
:duration
(= ?duration 1)
:condition
(and
(at start (connectivity-available ?ap ?ma1 ?t ?ma2 ?con))
(at start (application-ready-in-apache ?app ?ap ?ma1))
(at start (tomcat-working ?t ?ma2))
(at start (application-ready-in-tomcat ?app ?t ?ma2))
(at start (apache-working ?ap ?ma1))
(at start (machine-working ?ma1))
(at start (machine-working ?ma2))
(at start (not (machine-failed ?ma2)))
(at start (not (machine-failed ?ma1)))
(at start (apache-providing-service ?ap ?ma1 ?s))
(at start (application-requires-service ?app ?s))
(at start (apache-has-installed-module ?ap ?mo ?ma1))
```

```
)
:effect
(and
        (at end (application-ready-1 ?app ))

)
)
(:durative-action start-system-1a
:parameters (?ma1 - machine ?app - application
?ap - apache ?s - service)
:duration
(= ?duration 1)
:condition
(and
(at start (application-ready-in-apache ?app ?ap ?ma1))
(at start (apache-working ?ap ?ma1))
(at start (machine-working ?ma1))
(at start (not (machine-failed ?ma1)))
(at start (apache-providing-service ?ap ?ma1 ?s))
(at start (application-requires-service ?app ?s))
)
:effect
(and
        (at end (application-ready-1a ?app ))

)
)
(:durative-action start-system-1b
:parameters (?ma2 - machine ?app - application
?t - tomcat ?con - connector)
:duration
(= ?duration 1)
:condition
(and
(at start (tomcat-working ?t ?ma2))
(at start (application-ready-in-tomcat ?app ?t ?ma2))
(at start (machine-working ?ma2))
(at start (not (machine-failed ?ma2)))
)
:effect
(and
        (at end (application-ready-1b ?app ))

)))
```