A Planning Based Approach to Failure Recovery in Distributed Systems

Naveed Arshad
Computer Science Dept.
University of Colorado
Boulder, CO 80309
arshad@cs.colorado.edu

Dennis Heimbigner
Computer Science Dept.
University of Colorado
Boulder, CO 80309
dennis@cs.colorado.edu

Alexander L. Wolf
Computer Science Dept.
University of Colorado
Boulder, CO 80309
alw@cs.colorado.edu

ABSTRACT

Failure recovery in distributed systems poses a difficult challenge because of the requirement for high availability. Failure scenarios are usually unpredictable so they can not easily be foreseen. In this research we propose a planning based approach to failure recovery. This approach automates failure recovery by capturing the state after failure, defining an acceptable recovered state as a goal and applying planning to get from the initial state to the goal state. By using planning, this approach can recover from a variety of failed states and reach any of several acceptable states: from minimal functionality to complete recovery.

1. INTRODUCTION

Failure recovery in distributed systems is required to provide high availability to the users. 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 manual errors.

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 a way to bring the system back to a chosen working condition.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSS'04 Oct 31-Nov 1, 2004 Newport Beach, CA, USA Copyright 2004 ACM 1-58113-989-6/04/0010 $5.00.

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.

In the research described here, the goal is to automate the failure recovery procedure in a distributed system using AI planning. AI planning in combination with other techniques and tools can be a powerful mechanism for the failure recovery in distributed systems.

The main idea of our approach is taken from the Sense-Plan-Act (SPA) mechanism in control systems. *Sensing* is the detection of failure in a distributed system. There are sensors and detectors in the system that provides the information about the failure. *Planning* is a search mechanism to find the steps needed to go from the failed state to a normal working state. A planner is used for this purpose. And *Acting* is the execution of the plan on the actual system. An actuator executes the plan on the system. In this paper most of our discussion will be limited to the planning part of the failure recovery process.

2. PLANNING AND FAILURE RECOVERY

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, for each failure there are a number of ways to recover from it 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.

3. OVERVIEW OF AI PLANNING

We will give a brief overview of AI planning in this section. A detailed description of planning is out of scope of this paper. Interested readers should look at [3] for a detailed overview of automated 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 can not 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) [7]. We will use a pseudo version of this language to demonstrate our example. The AI community has constructed a number of planners [5] 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.

4. AN EXAMPLE SCENARIO

Before going into the details of how planning is helpful in recovering from failures, we look at a simple scenario of a distributed system.

In this scenario we have six non-client components deployed on five different machines. These components form a layered hierarchy based on inter-component dependencies. These six components are a web server, two servlet engines, two application server and a database. Moreover, there are clients connected to the web server. There are internal clients that support customer service and maintenance, and there are external clients supporting customers.

The distributed system is depicted in Figure 1. All the rectangles represent machines where instances of these components are working. All machines but one have a single instance of the component working. The instances of the servlet engine and the application server are not redundant. They have distinct sets of servlets and EJBs respectively. The application server also has a built-in servlet engine, but in order to provide better performance, separate servlet engines are installed. However, the built-in servlet engines in the application servers can be used if required.

The types of failures possible in this system are failures of the component or failure of the machine. In order to demonstrate our approach we assume that our sensors and detectors inform us about failure of machine 2. Further, we make the assumption that the failure of machine 2 is catastrophic and it can not be recovered in a reasonable amount of time.

The failure of machine 2 has a number of effects on the overall health of the distributed system. First of all the servlet engine on machine 2 also fails because of the explicit physical dependency. Second, the servlet engine fail-

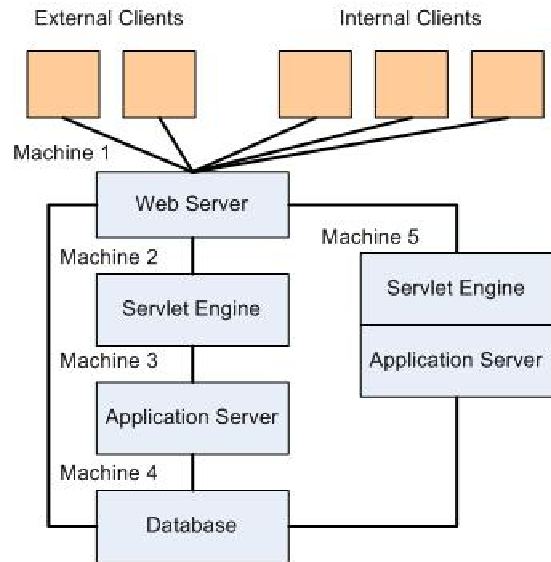

Figure 1: A Simple Distributed System Example

ure causes a ripple effect preventing the application server on machine 3 from delivering its functionality. Note that the application server is still running but it is unable to respond to any client requests because of its dependency on the servlet engine. Third, there is a set of clients that also lose their connection because of the failure. Other artifacts in the system continue to work as normal; the failure is not total.

The task of failure recovery is to bring the total functionality of the distributed system back online. The recovery process must not impact existing performance and functionality in the other parts of the system.

In order to recover the system there are many configurations that can be adopted. However, these configurations have different cost, time and resource implications. The goal of planning is to find the best alternate amongst these configurations. There are two general classes of recovery configurations that restore the system from failure taking into account the complete loss of machine 2.

1. Deploy the servlets that were in the failed engine into another servlet engine and connect that engine with the web server. There are two choices for the alternate servlet engine: either the servlet engine on machine 5, or the built-in engines associated with the application servers on machines 3 or 5.

2. Install a new instance of the servlet engine onto another machine (i.e. 1, 3, or 4) and then start it and connect it to the application server on machine 3.

The above failure recovery scenarios have different cost, time and resource implications. It may also be the case that we need a hybrid of the above approaches. This is especially true when there are some servlets that are critical and that needs to be back up online as soon as possible, while other servlets can wait for a certain period of time. Moreover, these target configurations require various steps to be carried out in a specific order before the system reaches a workable state. In order to make our discussion simple but concrete we will only consider the last set of scenarios that

```
Objects(applicationserver machine webserver servletengine)

Predicates
ServletEngineInstalled(servletEngine, machinename)
ServletEngineStarted(servletEngine)
ServletEngineWorking(servletEngine)
machineFailed(machinename)
ApplicationServerWorking(applicationServer)
WebServerWorking(webserver)
...
Functions
MachineRAM(machinename)
MachineStartTime(machinename)
ServletEngineInstallTime(servletEngine)
ServletEngineConnectTimeWithWS(servletEngine)
...
;; Actions
Start-Machine(machinename)
Duration (= (MachineStartTime(machinename))
Preconditions
            (not (machineFailed machinename))
effects
            machineStarted(machinename)
Install-Servlet-Engine(servletEngine machinename)
Duration (= (ServletEngineInstallTime(servletEngine))
PreCondition
            (> (MachineRAM(machinename) 512)
            (= (MachinePlatform(machinename) Unix)
            (= (MachineJDK (machinename) 1.4.2)
effects
            ServletEngineInstalled(servletEngine)

Start-Servlet-Engine (servletEngine)
Duration (= (ServletEngineStartTime(servletEngine))
PreConditions
            ServletEngineInstalled(servletEngine)
Effects
            ServletEngineStarted(servletEngine)

Connect-ServletEngine-AS(servletEngine, applicationserver)...)
Connect-ServletEngine-WS(servletEngine, webServer)...)
ServletEngineWorking...
```

Figure 2: A Subset of the Domain in pseudo-PDDL

assume that the lost servlet engine will be restarted on some other machine and that all the lost servlets will be deployed into that engine. The domain shown in Figure 2 can only be used to achieve such configurations. The connection restoration of the clients in the domain is not covered here.

5. THE USE OF PLANNING IN FAILURE RECOVERY

In this section we will see how planning can be used to recover from the above failure scenario. The three parts that are needed for carrying out AI planning are the domain, the initial state and the goal state.

5.1 Domain

The domain of the system is a static entity that is constructed before the distributed system goes into production, and it can not be changed during the construction of a plan. This is not a significant problem since the system architecture is unlikely to change rapidly, so the domain need not be changed unless the semantics of the system changes. It should be noted that the addition of new instances of existing components into the system usually will not affect the domain.

A subset of the domain for our example system is depicted in pseudo-PDDL in Figure 2. The domain description has three parts: *predicates*, *functions* and *actions*.

Actions generally correspond to the interfaces in present day ADLs [8]. Actions are the set of operations that can be executed on the system. Each action has a duration. The duration is defined as a function in the functions list. The numerical value of the duration is specified in the initial state. The preconditions of the actions are a set of logical expressions over the predicates and functions. In general all preconditions that involve a number are defined using functions and all preconditions that are boolean are defined using predicates. Predicates also serve the purpose of specifying the state of each component in the system (i.e., the equivalent of ground assertions in, say, Prolog). All predicates by default are false unless specifically asserted in the initial condition.

An action can only take place if the desired preconditions are fulfilled. If the preconditions are not fulfilled additional actions must be taken to fulfill those preconditions. Each action has a set of effects. The effects are represented in terms of functions and predicates also. The effects may change a boolean value of a predicate or change a numerical value in a function. The effects in general represent the new state of the system.

5.2 Dynamics of the Planning Process

The non-static elements of planning are the initial state and the goal state of the system. They are dependent on the individual scenario, so they must be constructed at runtime. Since a failure has a ripple effect in the system, a preprocessing step is required to compute the set of other components affected by the failure. After this preprocessing the initial and goal states are constructed and given to the planner to find a plan (a sequence of actions) to get from the initial state to the goal state.

The basic procedure for applying planning to a failure is as follows.

1. Check if a component has failed. This can be done either by polling of machines and components, or by some form of event notification.

2. For the failed component (including machines), calculate its ripple effect using a dependency model.

3. Construct the "system model" for each of the failed components. The system model is a set of predicates and functions describing the component's current state. The system model becomes part of the initial state for the planner.

4. Choose the goal configuration.

5. Invoke the planner with the domain, initial and goal states as input and obtain a plan in return.

6. Execute the plan for system recovery

Steps 1, 5, and 6 have been discussed elsewhere [1]. The focus here is on steps 2, 3, and 4.

5.3 Dependency Model

A dependency model is used to check the extent of damage after a failure. It is represented as a graph that determines the dependencies amongst the components. The dependencies also take into account the states of the component in which the dependency becomes effective. For example the applications server and servlet engine are dependent on each other because they need to be connected for the dependency to be effective. However, they can not be connected unless both of them are in a started state. If the connection is lost then both of them will change state and the dependency will lose its effectiveness. The dependency will still hold, however, and in order for the system to work normally the dependency must be restored. The dependency model also takes into account hardware dependencies such as the machine or network availability. This means that the hardware resources in the system are also part of the dependency model.

The dependency model is traversed to make a list of the components affected by the failure. For each component the state and present properties are calculated. This information classifies the components into three sets. 1) *Failed components* are the ones that have totally lost their functionality and must be restarted; 2) *Affected components* are the ones that are affected by the failed components and can not deliver any functionality, however, they are in a ready state and need not to be restarted; 3) *Normal components* are the ones that are not affected by the failure.

This information is used to develop a system model. This is a list of the components classified according to the three sets along with the properties of the components such as version and start-time, and along with the present state of the components.

This system model is then used to fill the initial state of the system (Figure 3). The critical thing to note is that if a planner is given more information, it will have a bigger search space, and this means it will take more time to find a plan. Therefore, it is necessary to construct an initial state that has the minimal set of information needed for reconfiguration. Each of the three sets of components will have differing amounts of information included in the initial state. For the failed components all the information is provided. For the affected components a smaller amount of information is provided and for the normal components very little information is provided.

5.4 Initial State

The initial state (really the current state) contains the list of all the components present in the system, their names, state and properties (asserted functions and predicates about the current state of the component). For instance the function *MachineRAM(machineName)* in the domain will specify the RAM available on a certain machine as *=(MachineRAM(machine1) 512))* in the initial state.

5.5 Goal State

The goal state is the desired configuration after the failure. There are two ways in which a goal state can be represented: *Implicit State* and *Explicit State*. In an implicit state the predicates that need to be true are specified in the goal state and it is up to the planner to find the best configuration possible and the best set of steps to reach that configuration. In an explicit state, however, an explicit configuration is

Initial State and Goal State

Objects

applicationserver1 – applicationserver...
servletengine2 – servletengine...
webserver – webserver
database – database
machine1 – machine...

Init

machineStarted(machine1)
machineFailed(machine2)
machineStarted(machine3)
..

= (machineRAM(machine1) 512)
= (machineRAM(machine3) 1024)
= (machineRAM(machine4) 1024)
..
= (machineJDK(machine1) 1.4.2)
= machineJDK(machine3) 1.3)
..
= (machinePlatform(machine1) Unix)
= (machinePlatform(machine3) windows2k)
..
servletEngineWorking(servletengine2)
applicationServerWorking(applicationserver2)
databaseWorking(database)
..

Goal

servletEngineWorking(servletengine1)
applicationServerWorking(applicationserver1)

Metric

Minimize Total-time

Figure 3: A List of Components and Machine, Initial State, Goal States and Metric in pseudo-PDDL

given and the planner's job is to select the best set of steps to reach that defined explicit configuration.

In the case of failure scenarios the state of the system is somewhat chaotic, so it may not be possible to have a explicit configuration and/or the resources to reach it. In failure scenarios, therefore, it is best to have an implicit configuration and let the planner decide the best alternate configuration. Remember that the goal is to bring the system back up online as soon as possible. The configuration chosen by the planner may be less than optimal but may be achieved in a relatively short span of time.

The implicit goal state consists of the truth values of the predicates that show the working state of the system. Therefore, the predicate *servletEngineWorking(servletengine1)* is the goal that will construct a plan to make this predicate true. While constructing the plan the planner takes care of the metric specified with the goal condition. If the metric is minimize total time, it will try to find a plan that minimizes total time.

5.6 Plan

A plan is a partially ordered set of steps to move from the failed configuration to the working configuration. The plan lays out the steps with the time they should begin. The timeline of the steps is mostly dependent on the dependency on other actions and/or availability of the resources.

In most planners one can specify how much time the plan-

ner has to find a plan. The plan found by the planner may not be the best, but it gives a plan that brings the system back online. If the planner finds a plan before the finishing time it gives out the plan but continues to try to find a better plan in remaining time. Usually the more time one gives to the planner the better it performs.

6. DISCUSSION

Planning can be used to automate the failure recovery in distributed system. However, the most difficult part of planning is defining a useful domain. Most of the time spent in designing a planning application involves making a thoroughly tested domain. To get better plans, the domain must be constructed taking all the semantics of the system into account. The domain itself is static and it is not easy to change it once the distributed system goes into production.

Failover capabilities are present in distributed systems in terms of network dispatcher and databases. The network dispatcher in a distributed system is used for the failover situations. However, in cases of disaster where most of the machines fail the network dispatcher can not do much to mitigate the failure scenario. Moreover, the network dispatcher itself may fail. Planning is useful mostly in disaster situations where most parts of the distributed system are not able to function.

Databases also often provide failover capability in a high availability mode. In this mode there are two instances of the database running concurrently. If the primary one fails for some reason the secondary one takes over. This approach is not geared towards the high availability mode of databases. We think that this approach is useful at the middleware level where the automated recovery of failure is still a problem.

Furthermore, for a distributed system where there is a *single point of failure*, the network dispatcher and databases failover mechanisms have limited use.

7. RELATED WORK

Failure recovery in distributed systems is related to the classical problem of dynamic reconfiguration in distributed component based systems [6, 11]. However, using planning for this purpose is relatively a new technique [1]. Moreover, recently there are planners developed specifically for deployment and reconfiguration [5].

Another area related to failure recovery is disaster recovery. A disaster is a critical failure in a distributed system which results from natural disasters etc. There are guidelines for recovery in such scenarios from NIST [9] and other companies like IBM (Tivoli Products) which outlines plans for disaster recovery. However, most of these plans are manual procedures.

Moreover, research has also been conducted on use of ADLs in dynamic runtime evolution of architecture which can be used in failure recovery also [10, 4, 2].

8. CONCLUSION

We have used planning successfully in the deployment and dynamic configuration of distributed systems [1]. In this work our focus was on the use of planning in the failure recovery scenarios of distributed systems. Based on our experiences we believe that planning can be used in automating the failure recovery in distributed systems, and that it will perform significantly better than existing manual or statically automated approaches.

9. ACKNOWLEDGEMENTS

This material is based in part upon work sponsored by DARPA, SPAWAR, and AFRL under Contracts N66001-00-8945, F30602-00-2-0608, and F49620-01-1-0282. The content does not necessarily reflect the position or the policy of the Government and no official endorsement is implied.

10. REFERENCES

[1] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. In *Proc. of the 15th IEEE Int'l Conf. on Tools with Artificial Intelligence*, pages 39–46. IEEE Press, November 2003.

[2] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proc. of the first workshop on Self-healing systems*, pages 27–32. ACM Press, 2002.

[3] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[4] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. Taylor. xADL: Enabling Architecture-Centric Tool Integration with XML. In *Proc. of the 34th Annual Hawaii Int'l Conf. on System Sciences (HICSS-34)-Volume 9*, page 9053. IEEE Computer Society, 2001.

[5] 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.

[6] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[7] D. Long and M. Fox. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research (JAIR), Special Issue on the 3rd Int'l Planning Competition*, 20(1):61–124, 2003.

[8] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Engineering*, 26(1):70–93, 2000.

[9] National Institute of Standards and Technology. *Contingency Planning Guide for Information Technology Systems.* (http://csrc.nist.gov/publications/nistpubs/800-34/sp800-34.pdf).

[10] R. Roshandel, A. van der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae : A System Model and Environment for Managing Architectural Evolution. *IEEE Trans. Software Engineering (Accepted for publication)*, 2004.

[11] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph-based Architectural (Re)configuration Language. In *Proc. of the 8th European Software Engineering Conf. held jointly with 9th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, pages 21–32. ACM Press, 2001.