Dealing with Failures During Failure Recovery of Distributed Systems

Naveed Arshad Dennis Heimbigner Alexander Wolf

CU-CS-1009-06

May 2006

University of Colorado at Boulder

Technical Report CU-CS-1009-06
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309–0430

Dealing with Failures During Failure Recovery of Distributed Systems

Naveed Arshad Dennis Heimbigner Alexander Wolf

May 2006

Abstract

One of the characteristics of autonomic systems is self recovery from failures. Self recovery can be achieved through sensing failures, planning for recovery and executing the recovery plan to bring the system back to a normal state. For various reasons, however, additional failures are possible during the process of recovering from the initial failure. Handling such secondary failures is important because they can cause the original recovery plan to fail and can leave the system in a complicated state that is worse than before. In this paper techniques are identified to preserve consistency while dealing with such failures that occur during failure recovery.

1 Introduction

Failure of the software and hardware components of a computer system is an expected, though undesirable, event. These failures may occur for many reasons. Some of the reasons include external attacks, internal faults, configuration mismatches, bugs, etc. Techniques to recover the components from failures have been available for a long time, but these techniques are mostly manual [11]. Automated failure recovery techniques are now being developed to materialize the notion of self recovery. Most of these techniques follow a general framework of sense-plan-execute [10].

One aspect of recovery that is rarely addressed, however, is the failure of components during the recovery process itself. The overall system is in an unstable condition during the recovery process, so there is no guarantee that other unaffected parts of the system will continue to work as normal during the recovery process. These failures can be detected at any time during the recovery process (Figure 1). Consequently, failures during each phase of the recovery process require specialized handling as determined by the phase properties. Moreover, once the system is recovered, it can not be assumed that the recovery process has recovered the system properly. There can be instances where the system seems to be recovered but it is not functioning as desired. Therefore, some form of acceptance test must be applied to the system to ensure its reliability.

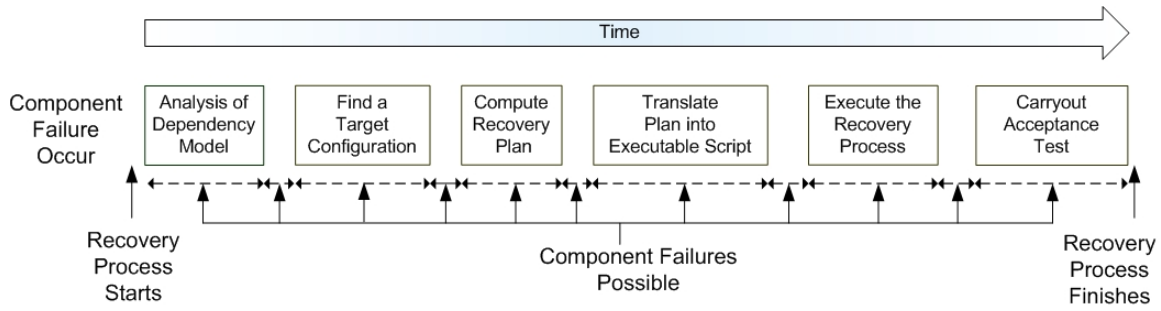

Figure 1: Failure detection that results in a decision to either continue or restart the recovery phase.

In the next section we give a brief overview of our approach to the recovery process in distributed systems. In subsequent sections, we describe the basic motivation of our approach of dealing with failures during recovery. From there we take each step in the recovery process and discuss what failures may occur in the system and how to handle them.

2 Failure Recovery Model

The failure recovery model we are using is a dynamic recovery model [12]. In a dynamic recovery model the handling of the failure is decided and applied at runtime. Several approaches to dynamic recovery are possible, however, our recovery model is based on three phases: Sense, Plan and Execute [2].

2.1 Sense

In the sense phase of the recovery process a monitor receives notification from the components about their health. These notifications are sent by probes inside the component. The notifications can be in the form of a regular heartbeat or in the form of a response to an explicit ping by the monitor. If the monitor suspects the component is behaving incorrectly, it puts the component in a suspected state and carry out some more tests to find out if the component has failed. If it determines that the component has failed it starts the planning process to find a plan to recover the failed component.

The sensing phase occurs continuously during the lifetime of the system; that is, the sensors report all failures no matter when they occur. This means that during the recovery process the monitor still continues to receive notification about the failure of components.

2.2 Plan

In the planning phase, the goal is to find a plan that recover the failed components while minimizing the effect on the other parts of the system. Planning has many sub-phases. The first sub-phase is to analyze a dependency model to determine the current components' state. The second sub-phase is to find a target configuration of the system so that system can be expected to start working correctly again if it reaches that target configuration.

The third sub-phase is compute a plan for recovery. Our approach uses an AI Planner [8] to compute this plan. During this planning sub-phase, the planner take as input a domain which specifies the semantics of the system. Additionally, it takes an initial state (i.e. the present (failed) state of the system) and it takes a goal (recovered) state of the system. Based on the semantics of the domain and the initial and goal state it computes a plan. A plan is a set of steps that take the system from the initial state to the goal state.

The fourth and final sub-phase is to translate the plan into a recovery script that can be executed to recover the application into the target configuration.

2.3 Execute

The execute phase takes the recovery script and executes it. At the end of the execution, the system should be restored to correct operation. After the recovery completes an acceptance test is applied to the system to ensure its reliability.

3 Problems and Assumptions

The problems in dealing with failures during failure recovery stem from the arbitrary nature of failures. Failures can be detected at any time during the lifetime of the system. Even when the system is recovering from a failure, other failures may be detected. Therefore, the failure recovery system must be able to handle failures at any time.

Moreover, when a system recovering from failure, it is in an unstable state. Therefore, if further failures are not handled properly then they may cause the system to go into an inconsistent state, and it may be substantially harder to recover from this state. Furthermore, simultaneously recovering two or more components without a global recovery plan may create race conditions. The components may be trying to get hold of the same resource or doing operations to undo each other's effect. Thus, the failures in the system must be handled systematically. However, before discussing the details of such approach there are some assumptions that are necessary for our approach to work.

- *No False Positives.* The first assumption is that no false positives occur in the system. False positives occur when a component temporarily malfunctions and stops the component from sending any heartbeat or makes it unable to respond to any ping. After some period of time the component may start functioning again and may send the periodic heartbeat. Packet loss and other network malfunctions also

contribute to this problem. In our present model we assume that the components have a fail-stop behavior and if they fail they do not start working again unless a recovery process is executed on them. We also assume that the network is reliable and efficient and packet loss will be hidden at the network layer.

- *Perfect Recovery.* The second assumption is that the recovery process is perfect. By this we mean that given a recovery script, the recovery is executed without any failures, although the recovery plan may be flawed. This assumption helps to scope the problem by allowing us to focus on the failure of other components during the recovery process: failures that can alter the path of the recovery process.

4 Dependency Model

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. This model is represented in form of a dependency graph having different edges to specify the kinds of dependencies. Each component of the system is a node in the dependency graph.

4.1 Kinds of Dependencies

There are two kinds of dependencies in the dependency graph: hard and soft.

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 2 by a solid line in the 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 2 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.

4.2 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 ($\mathcal{F}$).

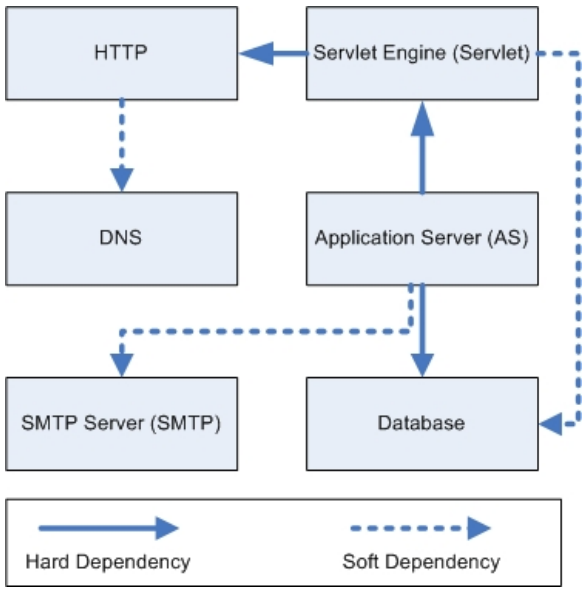

Figure 2: Dependency graph of system components.

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 $\mathcal{W}$.

5 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 2 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 Planning

Planning has various sub-phases as shown in Figure 1. 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 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.

6.1 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 1 provides a summary of the process.

6.1.1 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.

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

Table 1: A summary of what happens to the recovery process when more components fail.

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.

6.1.2 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.

6.1.3 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.

7 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.

7.1 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 $\mathcal{W}$.

7.1.1 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.

7.1.2 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.

7.1.3 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 [3].

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%.

$$Yield = \frac{tests\ completed}{tests\ offered} \tag{1}$$

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%.

$$Harvest = \frac{components\ accessed}{total\ number\ of\ components} \tag{2}$$

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.

8 Related and Future Work

Most of the literature in the fault tolerance and autonomic computing does not take into account the fact that failures are possible during the healing process. Therefore, this work is building upon the previous work [6, 12, 7] in failure recovery and adding ways to handle failures during failure recovery.

However, failures during failure recovery poses a tough problem because of the complexities involved. In this paper we have presented some techniques based heavily on the dependency model of the system. There is some work on dependency formalization [1, 9, 5]. Our goal is to extend this work and make a more rigorous dependency model. We will use this model not only in the failure recovery but also in the inter and intra component configurations. Moreover, finding and formalizing dependencies are required in the techniques presented in this paper. Even if the dependencies are specified, there could be some hidden dependencies present in the system. There is also work in finding hidden dependencies in the system [4]. Therefore, we believe that there is a promise in using the dependency model for dealing with failures during failure recovery.

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.

References

[1] S. Alda, M. Won, and A. 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.

[2] N. Arshad, D. Heimbigner, and A. 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.

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

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, June 2002.

[5] L. Cox and H. 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.

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

[7] S. George, D. Evans, and L. 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.

[8] 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.

[9] A. Keller and G. Kar. Dynamic dependencies in application service management, 2000.

[10] J. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, and P. Devanbu. The willow survivability architecture, 2001.

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

[12] J. Park and P. Chandramohan. Static vs. dynamic recovery models for survivable distributed systems. In *HICSS*, 2004.