CRASH-ONLY SOFTWARE AND MICROREBOOT: A DESIGN AND TECHNIQUE FOR ACHIEVING HIGH AVAILABILITY IN FREQUENTLY-FAILING SOFTWARE SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

George M. Candea
September 2005

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

© Copyright by George M. Candea 2005
All Rights Reserved

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Armando Fox   Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David R. Cheriton

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the University Committee on Graduate Studies.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Abstract

Application-level software failures are a dominant cause of outages in large-scale software systems, such as e-commerce, banking, or Internet services. The exact root cause of these failures is often unknown and the only cure is to reboot, often at the cost of nontrivial service disruption or downtime, even when clusters and failover are employed. Our thesis is that, although large-scale software systems are unreliable, structuring them for fast, minimally-disruptive recovery is a cost-effective way to make them highly available.

This dissertation defines the crash-only design, a set of principles for building large-scale programs that crash safely and recover fast. We describe the microreboot mechanism, by which fine-grained components of crash-only systems are recovered through restart at the first indication of failure. We applied the crash-only design and microreboot technique to a satellite ground station and an Internet auction system; without fixing any bugs, microrebooting recovered most of the same failures as process restarts, but did so more than an order of magnitude faster and with an order of magnitude savings in lost work, reducing overall unavailability by a factor of 50.

The fast, minimally-disruptive nature of microrebooting makes several failure management policies cost-effective, policies that would otherwise be prohibitively expensive in terms of incurred downtime. First, we show that failures can be avoided at low cost by preventively microrebooting components, thus rejuvenating applications with minimal downtime. Second, we show that microrebooting at the slightest hint of failure (without engaging in diagnosis) improves availability even when failure detection is prone to false positives. Finally, we demonstrate that microreboot-based recovery can be hidden from end users via transparent request retries, improving availability without change in end-user-perceived service quality.

The crash-only/microreboot approach is in keeping with a minimalist philosophy of system design, in which simpler recovery mechanisms are preferred to complex ones – by casting most failures as reboot-curable problems, we simplify recovery, making it more prompt and effective, while being less disruptive to end users. We conclude that the combination of crash-only software and

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

microrebooting provides a better cost/dependability tradeoff compared to the traditional approach of aiming for correct code and supporting diverse recovery mechanisms.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Acknowledgments

My PhD career has been an amazing experience; even if I were to not receive my degree, I would still do it all over again.

I would like to thank Armando Fox, the perfect PhD adviser for me. He masterfully turned this young graduate student into a confident researcher; Armando nudged me back into the right path when I strayed, and stepped out of the way when I wanted to run on my own. He never lost faith and was a constant source of encouragement.

David Patterson, my de facto secondary adviser, kept me from running down rat holes and pushed me into pursuing the highest-yield ideas. Jim Gray was the one who, back in 1999, advised me to take the low-MTTR road to high availability; he has been a real mentor ever since. David Cheriton, who is many years ahead of everyone else, was the first to believe and support my reboot-based approach to recovery from the early days of Medusa, when it was all just a fuzzy bunch of half-baked thoughts. Thank you, Mendel Rosenblum and Martin Rinard, for your patience, advice, and encouragement.

The results of the research described here, as well as the experience itself, would not have been the same without my close collaborators: Shinichi Kawamoto, Emre Kiciman and Jamie Cutler. Members of the ROC project have suggested many great ideas over the years and helped me winnow my own; thank you Aaron Brown, David Oppenheimer, and all the other ROCers. All my collaborators have been crucial to the realization of this dissertation: Mauricio Delgado, Greg Friedman, Yuichi Fujiki, Pedram Keyani, Steve Zhang, Rushabh Doshi, Priyank Garg, Rakesh Gowda, JP Schnapper-Casteras, Mark Brody, and Tom Martell. Thank you, my fellow SWIG mates – Laurence Melloul, Andy Huang, Ben Ling, and Shankar Ponnekanti – for numerous discussions, feedback sessions, and draft readings. My students in CS444 have been teachers to me, although they probably don't know it. Thank you Sarah Weden for making administrative life so much easier. Andy Kacsmar has often lent a helpful hand in fixing/installing my computer hardware, although it was not his job to do so.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

My roommate and friend, Scott Brave, has been an example of intelligent goodness, calm, and ethics. Thank you, Katerina Argyraki, for showing me how all complicated things are in effect really simple. Karate has kept me sane throughout my graduate years, and I thank all my senseis and senpais for yanking fear out of me and giving me strength.

Most of the credit, however, is due my parents. They live their life to make me happy, and they have given me everything I needed to find my way and succeed. They have always been an inspiration, an island of relief, and an unshakable source of support. Mamuţa şi Tatuţa, this dissertation is dedicated to you.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Contents

Abstract v

Acknowledgments vii

1 Introduction 1

1.1 The Problem: Unreliable Software 1

1.1.1 Software-induced downtime 2

1.1.2 Why is software unreliable ? 2

1.1.3 Will software become more reliable? 5

1.2 High Availability despite Software Failures 7

1.2.1 Mitigating unreliability through fast, minimally disruptive recovery 7

1.2.2 Rebooting – a pragmatic recovery technique 8

1.2.3 Benefits of reboot 8

1.2.4 Drawbacks of reboot 9

1.2.5 Benefits without drawbacks 10

1.3 Thesis and Contributions 11

1.4 Outline 12

2 Related Work 13

2.1 Preventing Software Failures 13

2.1.1 Better software 13

2.1.2 Containing failure propagation 14

2.1.3 Software rejuvenation 16

2.2 Detecting failures 16

2.3 Tolerating software failures 18

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

2.3.1 Redundancy 18
2.3.2 Saving state 20
2.3.3 Careful state updates 21
2.3.4 Making recovery fast 22
2.4 Chapter Summary 23

3 Background **25**
3.1 Medusa: An Execution Platform for UNIX Programs 25
3.1.1 Overview 25
3.1.2 Lessons 27
3.2 Mercury: Control Software for a Satellite Ground Station 28
3.2.1 Overview 28
3.2.2 Lessons 30
3.3 Chapter Summary 31

4 Crash-Only Software **33**
4.1 Reboot-based Recovery 34
4.1.1 Rebooting works around Heisenbugs 34
4.1.2 Rebooting returns system to a known state 34
4.1.3 Rebooting is simple and unequivocal 35
4.1.4 Rebooting is effective against poorly understood failures 35
4.1.5 Rebooting can result in data loss/corruption 36
4.1.6 Rebooting can induce lengthy recovery 36
4.2 Why Crash-Only Design ? 37
4.3 Principles of Crash-Only Design 40
4.3.1 Fine-grained isolation 41
4.3.2 State segregation 42
4.3.3 Inter-component decoupling 44
4.3.4 Component/request decoupling 45
4.3.5 Component/resource decoupling 45
4.4 Discussion 46
4.5 Chapter Summary 48

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

5 **Microreboot-based Recovery** 51
5.1 The Microreboot Mechanism . . . 51
5.1.1 Platform-level support . . . 53
5.1.2 State stores . . . 53
5.1.3 Correctness of microrebooting . . . 54
5.2 Microreboot-based Recovery Policy . . . 57
5.2.1 Recovery groups . . . 57
5.2.2 Recursive microrebooting . . . 58
5.2.3 Tradeoffs . . . 59
5.3 Chapter Summary . . . 61

6 **Prototype and Experimental Setup** 63
6.1 Overview of J2EE . . . 64
6.2 Synergies with Crash-Only Design . . . 65
6.2.1 Component management . . . 66
6.2.2 Decoupling . . . 66
6.2.3 State segregation . . . 67
6.3 Testbed . . . 68
6.3.1 A microreboot-enabled application server . . . 68
6.3.2 EBid: A crash-only application . . . 72
6.3.3 State segregation in EBid . . . 72
6.3.4 Session state storage . . . 73
6.3.5 Recovery manager . . . 75
6.4 Failure Detection and Localization . . . 76
6.4.1 Failure detection . . . 77
6.4.2 Score-based fault localization . . . 78
6.4.3 Inferring application structure . . . 78
6.5 Client Emulator . . . 79
6.6 Action-Weighted Throughput . . . 79
6.7 Chapter Summary . . . 81

7 **Evaluation of the Microreboot Recovery Mechanism** 83
7.1 Recovery Effectiveness . . . 84
7.2 Recovery Efficiency . . . 87

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

7.2.1 Faster recovery 89
7.2.2 Less functional disruption 90
7.2.3 Less lost work 91
7.2.4 Summary 92
7.3 Microrebooting in a Cluster 92
7.3.1 Conserving session state during failover 92
7.3.2 Preserving response time during failover 94
7.3.3 Summary 96
7.4 Performance Impact 96
7.5 Chapter Summary 97

8 Exploration of Microreboot-based Recovery Policies 99
8.1 Lax Failure Detection 100
8.1.1 Cheap recovery allows longer detection times 100
8.1.2 Cheap recovery tolerates occasional mistakes 101
8.1.3 Summary 102
8.2 Autonomous Recovery 103
8.2.1 Application-generic failure detection 103
8.2.2 Recovering autonomously 104
8.2.3 Summary 106
8.3 Alternative Failover Schemes 106
8.4 Multi-Tier Recovery 107
8.5 Microrejuvenation: Preventing Failures Caused by Resource Leaks 109
8.5.1 Resource leaks lead to failure 109
8.5.2 Rejuvenation prevents catastrophic failure 110
8.5.3 Microrejuvenation 111
8.5.4 Summary 113
8.6 User-Transparent Recovery 113
8.7 Chapter Summary 115

9 Limitations and Challenges 117
9.1 Results Proven in Java Environments 117
9.2 Microreboots Only Cure Transient Failures 118
9.3 Fine-grained Recovery Requires Fine-grained Workloads 118

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

9.4 State Segregation Requires Discipline 119
9.5 Achieving Good MTTF/MTTR Balance is not a Rigorous Process 120
9.6 Componentizing Legacy Software is Difficult 120

10 Conclusions **123**

A Automatic Failure-Path Inference **125**
A.1 Approach 126
A.1.1 AFPI algorithm 126
A.1.2 Modifying JBoss to enable AFPI 128
A.2 Experiments 129
A.2.1 Suitability of injecting Java exceptions 129
A.2.2 F-maps compared to existing structures 131
A.2.3 Fault-class-specific f-maps 135
A.2.4 Correlated faults 136
A.2.5 Performance overhead 136
A.2.6 Weaknesses 137
A.3 Summary 137

B Roadmap to Representative Publications **139**

Bibliography **141**

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

List of Tables

1.1 Trends in causes of outages for high-end Tandem NonStop systems [Gra90]. 6

4.1 Duration of clean vs. crash reboots. 37

6.1 Client workload used in evaluating microreboot-based recovery. 79

7.1 Recovery from injected faults: worst-case scenarios. 86
7.2 Average recovery times under load, at various granularities. 90
7.3 Requests exceeding 8 sec during failover under doubled load. 96
7.4 Performance comparison. 97

8.1 Masking microreboots with HTTP/1.1 `Retry-After`. 114

A.1 Exception types used in AFPI experiments. 131
A.2 RMI experiments evaluating the conversion of real faults into Java exceptions. . . . 131

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

List of Figures

1.1 Code base evolution for Microsoft Windows and Linux. 4

3.1 Mercury software architecture 29

5.1 A crash-only application running atop a microreboot-enabled execution platform. . 52
5.2 Pseudocode for a generic microreboot facility. 53

6.1 Architectural diagram of a J2EE application environment. 64
6.2 Conceptual architecture of our J2EE prototype. 68
6.3 Pseudocode for the JBoss implementation of microreboot. 69
6.4 Sample code using the SSM/FastS session state interface. 74
6.5 An f-map for eBid, obtained with AFPI. 78

7.1 Impact of recovery on end users: process restart vs. microreboot. 88
7.2 Functional disruption, as perceived by end users. 91
7.3 Conceptual architecture of the cluster testbed. 93
7.4 Failover under normal load. 94
7.5 Failover under doubled load. 95

8.1 Relaxing failure detection: $FP_{\text{diag}} = 0$ constant, while T_{det} varies. 101
8.2 Relaxing failure detection: $T_{\text{det}} \approx 0$ constant, while FP_{diag} varies. 102
8.3 Autonomous recovery 105
8.4 Autonomous recovery: zooming in on time interval [5:32, 6:00]. 105
8.5 Abstract view of the design space for software recovery. 108
8.6 Memory leaks induce failure. 110
8.7 Full rejuvenation: T_{aw} and level of available memory. 111
8.8 Microrejuvenation: T_{aw} and level of available memory. 112

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

A.1 F-map for Petstore: deployment descriptors (top) vs. AFPI (bottom). 132
A.2 F-map for RUBiS: deployment descriptors (top) vs. AFPI (bottom). 134
A.3 Petstore f-map based exclusively on application-declared exceptions. 135

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 1

Introduction

Computer systems are being used in increasingly larger segments of everyday activities, propelled by exponential increases in performance and functionality, compounded by exponential decreases in cost. Software is making its way into every aspect of our lives, from transportation, communication, and financial systems to cellular phones and entertainment. Dependability requirements previously conceived only for small, isolated, special-purpose systems are now expected of many large, interconnected, rapidly-evolving systems. These factors make building, deploying, and managing *dependable software systems* an urgent task.

1.1 The Problem: Unreliable Software

Software failure is a threat to our life and productivity. In August 1997, a software defect in the Minimum Safe Altitude Warning System at Guam's airport led to the crash of Korean Airlines flight 801, killing 225 people [Nat98]. In February 1991, a Patriot missile defense system failed to track and intercept an incoming Scud missile during Operation Desert Storm, due to an error accumulation in its control software [Off92], resulting in 28 dead and 98 wounded American soldiers. A bug in General Electric's XA/21 energy management system contributed in August 2003 to the scope of the worst power outage in U.S. history, that spread across the Northeastern U.S. and Canada [Pou04], affecting 50 million people and disrupting transportation, water, and many other services. At EBay, a popular online auction service, a flaw in the Sun Solaris operating system led to database file corruptions that brought the service down for 22 hours in June 1999; EBay's direct costs were estimated at $3-5 million, and EBay's stock price dropped by 26%, erasing $4 billion in market capitalization [FP02].

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

In terms of direct costs, the National Institute of Standards and Technology (NIST) estimated in 2002 that software faults cost the U.S. economy $59.5 billion annually, which represents 0.6% of the Gross Domestic Product [NIS]. These numbers suggest that unreliable software is emerging as perhaps the single most significant problem faced by computer systems today.

1.1.1 Software-induced downtime

The goal of this work is to bring higher availability to software systems. Other important dependability concerns, such as security and safety, are outside the scope of this dissertation.

A dominant factor affecting availability of large-scale systems is their software. When a computer system fails, we generally say it is "down," and the time period when it is down contributes to its overall "downtime." In all large-scale, well-managed computer systems, failures are eventually analyzed and diagnosed; their root cause generally falls in one of three categories: hardware failure, software failure, or human error. Based on a scarce supply of public failure surveys for commercial systems, we conservatively estimate that software failure causes approximately 40% of outages in large-scale, well-managed commercial systems[1] (confirmed by [Woo95] for high-end transaction processing servers and by [CC02] for systems in general). When software-induced outages occur, their effects are compounded by the fact that a large fraction of bugs that manifest in production systems have no fix available at the time of failure; one study found this fraction to be 80% [Woo03].

Given sufficient time, software can mature and become more reliable. This is how, for example, the U.S. public switched telephone network (PSTN) is able to provide its legendary high availability. Kuhn [Kuh97] found that only 14% of the PSTN's outages between 1992-1994 were caused by software, coming in third after human error (49%) and hardware failures (19%). This seems to suggest that thorough design reviews and extensive testing could eventually improve the dependability of software systems single-handedly; we dispel this myth in the following section.

1.1.2 Why is software unreliable ?

Experience suggests that there is a significant limitation to how free of bugs a program can be. Researchers and engineers have improved programming languages, built powerful development and testing tools, designed metrics for estimating and predicting bug content, and assembled careful development and quality assurance processes. In spite of all these, we've seen that software is still

[1] Some of the important factors that can cause this figure to vary include software quality, system architecture, quality of system management and administration, etc.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

far from perfect. Moreover, NIST estimates that 2/3 of software bugs that manifest in deployed systems could not have been readily caught by better testing processes [NIS]. Three phenomena conspire to limit the effectiveness of traditional approaches to software reliability: code evolution, unforeseen usage scenarios, and inadvertently deceptive abstractions. These three factors prevent software vendors from being able to guarantee a program of reasonable size will run as expected once deployed at the customer's site.

Change is anathema to dependability or, in engineering parlance, "if it ain't broke, don't fix it." Only in a software system that evolves very slowly is it possible to control the effects of change, and to maintain or improve software quality. Case in point: the space shuttle software consists of half a million lines of mostly Ada code and, as of 1997, its last three releases manifested one bug each, with the last 11 versions totaling only 17 bugs [Fis97]. Such reliability, however, comes at the expense of evolution: upgrading the space shuttle software to use GPS-based instead of land-based navigation was a major undertaking. The change involved only 1.5% of the code, yet simply formalizing the required specifications into 3,300 lines of PVS code[2] took 4 person-months [CV98], followed by an even longer development and test cycle. This is no surprise, considering that, prior to every flight, the team's senior technical manager must sign a document certifying to NASA that the software will not endanger the shuttle [Fis97].

Such rigidity would threaten the very existence of most software systems in use today and in the near future, mainly because of customer demands and time-to-market pressures. This is true of the whole software stack – operating systems, applications, and software services – both commercial and open-source. Consider, for instance, the rate of growth in two of today's popular operating systems (figure 1.1). The last 11 major versions of open-source Linux expanded the code base by two orders of magnitude over 10 years, while 7 major releases of Microsoft Windows increased its code base by more than one order of magnitude. The evolution due to addition of new code is further compounded by extensive changes to existing code, resulting in the thousands of bugs that lurk within Windows and Linux.

Most infrastructure service operators we have interviewed speak of reliable "change management" as a seemingly insurmountable challenge. The rate of change in packaged applications turns out to be higher than in operating systems, and even higher in online Web services. For example, it is common for developers at Yahoo, a popular Web directory and portal, to deploy new code into production on a weekly basis [Man01]. When systems change at a faster rate than the rate at which

[2] PVS [ORSvH95] is a specification and verification system consisting of a specification language, tools and a theorem prover; it enables mechanized use of formal methods.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

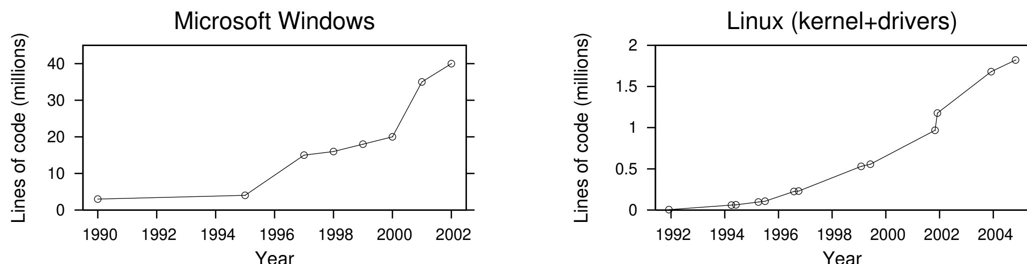

Figure 1.1: Code base evolution for Microsoft Windows and Linux.

development processes and tools can reduce the number of bugs per line-of-code, the net effect is that each new release introduces more bugs than it fixes.

Diverse execution environments and scenarios constitute the second factor that limits software quality. Returning to the space shuttle example, we observe that its software has the advantage of supporting only one platform and only one customer. In contrast, most software today must interact with a variety of devices, support a variety of configurations and uses, as well as be combined with various other third-party software. Even if a system's code base did not change at all, a new execution environment or scenario would unavoidably exercise a code path that had never been tested before, bringing out latent bugs. Even if testing all paths through a program was possible, testing all imaginable execution environments and the ensuing interactions would not. The more complex a software product, the more difficult it is to understand and predict its behavior in production, making quality assurance difficult. For example, in 2003, the Oracle 9i database server was subjected to a battery of 60,000 tests after development, yet it did not pass all of them prior to release – a fraction of these tests still failed, because the bugs were dependent on the testers' environment, making them difficult to reproduce, or were too expensive and/or risky to fix. This is true of all commercial software of nontrivial size.

Inadvertently deceptive abstractions provide the third impediment to software quality. As higher-level programming languages increase the level of abstraction, they hide from programmers the operating details of the underlying runtime environment. However, it is very difficult to completely specify the behavior of abstractions and to correctly implement them. For example, Java provides "care-free" memory management through the use of garbage collection. Yet, nonchalant reliance on this feature in server software can lead to failure: as memory usage increases, the garbage collector starts reclaiming objects; doing so during a load spike can result in the system thrashing and collapsing under load (one such instance is described in [Gri01]). Thus, writing robust server software in Java requires a thorough understanding of how the underlying Java virtual

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

machine (JVM) manages resources, an understanding that is not frequent among Java programmers. What's worse, when a JVM implementation or configuration switches between incremental and non-incremental garbage collection, previously robust code can become unpredictable. This is an example of how the memory abstraction in Java can be (inadvertently) deceptive.

While abstractions allow programmers to deal with new orders of complexity and to speed up development (e.g., GUI and network programming is considerably simpler today than two decades ago), they can also increase the incidence of bugs caused by a misunderstanding of the underlying layers. It is therefore not surprising that, of the bugs that manifest after deployment of commercial software in enterprise-scale systems, the lion share is held by Heisenbugs, race conditions, resource leaks, and environment-dependent bugs [CJ02, Rei04]. The risks introduced by deceptive abstractions are further compounded by the fact that apparent ability to build more complex systems drives customers to demand ever higher levels of (shabby) functionality.

1.1.3 Will software become more reliable?

Having identified code change and growth, diverse execution environments, and inadvertently deceptive abstractions as three of the factors that limit software reliability, we will now project existing trends into the future and explore whether these problems will become more or less challenging.

The very essence of software is its ability to be morphed at a moment's notice – change is fundamental to software. Future code evolution and growth is inevitable, driven by continuing feature pressure and the ability of hardware devices to accommodate increasingly larger footprints and perform more computation. Software products will constantly adapt to meet competitive demands and the most successful products will be those built on platforms designed to accommodate change. Resisting growth and change is not the path to higher dependability, as has been eloquently demonstrated by software running the Internet. Even the most closely guarded infrastructures will not be able to escape massive change. For example, failure reports for the U.S. public switched telephone network reveal that software-induced downtime went from 15 million customer-minutes/month in 1992-1994 [Kuh97] to 155 million customer-minutes/month in 2000 [Enr02]. This order of magnitude increase is explained by the rising number of (software-based) features offered by the telecom industry during that decade, as well as software changes made to handle the Y2K problem[3].

[3]The "Year 2000 Problem" resulted primarily from the use of only two digits to represent a year (e.g., "99" instead of "1999"). This flaw turned into a major fear that critical industries (electricity, financial, telecom, etc.) and government functions would stop working at 12:00 AM, January 1, 2000 (or 1/1/00). This fear was fueled by extensive press coverage and speculation, as well as copious official corporate and government reports. As year 2000 approached, companies and organizations worldwide invested heavily in checking and upgrading their computer systems.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

The diversity of execution environments will continue increasing as well, as the number of devices relying on software goes up and infrastructure services become more complex and more distributed. More software will be needed in more places. The commoditization of hardware will continue forcing software to adapt, even in the most specialized of systems. For example, Tandem NonStop systems are some of the most reliable servers in the world and run the vast majority of the world's securities trades, ATM transactions, and credit card transactions, as well as the US public telephone system, emergency-911 and various cellular networks. Yet, a 5-year survey [Gra90] (summarized in table 1.1) revealed that the fraction of Tandem outages due to software almost doubled over the period of the study. Gray [Gra90] explains this phenomenon by the fact that, during the 5-year period, Tandem's software tripled in size to add support for 3 new processor families and a variety of new peripherals, as well as to accommodate expanded functionality; in the meanwhile, hardware became more reliable and engineer-friendly, thus reducing the incidents of human error.

Cause of outage	Fraction of all outages		
	1985	**1987**	**1989**
Software failure	34%	39%	62%
Hardware failure	29%	22%	7%
Maintenance and operations	27%	25%	20%
Other	10%	14%	11%

Table 1.1: Trends in causes of outages for high-end Tandem NonStop systems [Gra90].

Finally, as more software will be needed in more places, there will be a need for more programmers that can prototype software faster – prime candidates for victims of inadvertently deceptive abstractions. Recent trends confirm our belief that increasingly more programmers will flock to the languages and tools that allow shorter development cycles and have easier learning curves. For instance, looking at the one-year changes since November 2003, we find that the world-wide availability of skilled engineers, courses and third party tools has increased most rapidly for Visual Basic, PHP, Delphi, Python [Tio04]; this group of 4 claims almost one third of the programming language space today. To make programming easy, these languages hide from the programmer resource management, synchronization control, and other underlying details. We believe this will eliminate some types of bugs, while introducing others that are more difficult to debug. Compounding this effect, shorter development cycles will likely shorten test cycles and pressure engineers into hastier designs.

The answer to this section's title is most likely *no* – software will not become more reliable in

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

the near future, due to sustained code growth and change, variety of execution environments, and inadvertently deceptive abstractions. In light of the evidence, we believe bug densities will increase, leading to software that fails more frequently.

1.2 High Availability despite Software Failures

To achieve high availability in the face of high failure probability, we focus on designing software systems such that they recover fast and reduce the impact of failure and recovery. In the limit, a system that instantly recovers from every fault is 100% available. In this dissertation, we show how to use fine-grained rebooting at the sub-process level to improve availability of systems by 1-2 orders of magnitude. The rest of this section argues for the benefits of fast, universal, minimally disruptive recovery in general, and then describes microrebooting, our proposed recovery technique.

1.2.1 Mitigating unreliability through fast, minimally disruptive recovery

Availability of a system is generally an expression of the system's readiness to deliver service, and can be expressed as a ratio between the system's mean-time-to-failure (MTTF) and its mean-time-to-recovery (MTTR):

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

MTTF is an expression of the system's *reliability*, because it describes how often the delivery of correct service will be interrupted by a failure [Lap91]. MTTR describes the ability of the system to return to correct functionality once it has failed.

The opposite of availability ("system up") is unavailability ("system down"):

$$\text{Unavailability} = 1 - \text{Availability} \approx \frac{\text{MTTR}}{\text{MTTF}} \qquad \text{(assuming MTTF} \gg \text{MTTR)}$$

Higher availability does not require higher reliability. As the above equation illustrates, one can reduce unavailability by a factor of n by increasing MTTF n-fold (i.e., by improving reliability n times). However, the exact same effect can be achieved by reducing MTTR n-fold (i.e., by speeding up recovery n times). The premise of our work is that improving MTTF has reached a point of diminishing returns: reducing unavailability by an order of magnitude through a ten-fold increase

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

in MTTF is more difficult and expensive than reducing MTTR by a factor of ten. This motivates our focus on improving MTTR.

There are two ways to reduce time-to-recover: First, one can preserve the current approach to recovery, but engineer the system to perform that recovery faster. Second, one can reduce the scope of recovery, thus making recovery faster *and* less disruptive. We choose the latter approach, and introduce the notion of *microrecovery* – recovery confined to the individual component that is faulty, instead of the entire system. In this dissertation we present in detail a reboot-based form of microrecovery, with the understanding that many of the resulting techniques and lessons are more widely applicable.

1.2.2 Rebooting – a pragmatic recovery technique

Rebooting is a simple, practical and effective approach to managing failure in large, complex systems; it is an approach that accepts bugs in applications as facts to be coped with, instead of viewing them as problems that can be eliminated. The results of several studies [SC91, Gra86, MG95, Cho97] and experience in the field [Bre01a, Pal02, Lev03] suggest that many failures can be successfully recovered by rebooting. Not surprisingly, today's state of the art Internet clusters provide facilities to circumvent a faulty node by failing over, rebooting the failed node, and subsequently reintegrating the recovered node into the cluster.

Rebooting can often take the form of just an application restart. Large-scale Internet software fails mostly due to application-level bugs [Mit04, Cho03, Pal02, Rei04], unlike desktop systems, which are still plagued by failures at the OS level, such as bad drivers [SBL03]. Most practitioners agree that hardware and operating system availability is a "solved problem" in enterprise-scale software, at least in comparison to application-level availability. Platforms (such as hardware, operating systems, and application servers) are subject to much less change than applications and are subject to considerably more testing. Unlike desktop systems, the nodes of today's Internet service clusters support few devices/drivers, have relatively simple configurations, and have been extensively tested prior to deployment. We therefore fold application process restarts into the reboot category, because they often are as effective as operating system reboots.

1.2.3 Benefits of reboot

Rebooting has three core benefits.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

First, rebooting scrubs volatile state that has potentially become corrupt. A bad pointer, a deadlock surrounding a set of mutexes, an accumulated computation error are all examples of volatile corruption that would be cleaned up by rebooting. Rebooting reclaims leaked resources and does so decisively and quickly, because mechanisms used to effect the reboot are simple and low-level: virtual memory hardware, operating system processes, language-enforced mechanisms, etc. Should an application leak memory or file descriptors, they will all be reclaimed upon restarting that application's process.

Second, rebooting returns an application to its start state (or at least a well-known state), which is the best understood, most thoroughly debugged state of the program. Whenever a program starts up, it begins in its start state, so this is the most frequently visited state during development, testing, and operation.

Third, rebooting improves MTTR by saving on diagnosis time. When failure strikes an Internet system, operators cannot always afford to run real-time diagnosis; instead, they focus on bringing the system back up by any means possible, and do the diagnosis later. Experienced operators realize that there is a large opportunity cost in taking several minutes under fire to decide whether a reboot would or would not cure the failure, whereas a minute-long reboot would answer the question much sooner. Rebooting is simple, so implementing and automating a recovery policy based on rebooting is perhaps the easiest and simplest of all recovery alternatives. The fact that rebooting an application requires nothing more than a process kill and start is unique among recovery strategies.

Rebooting is in some sense a universal form of recovery, since a failure's root cause does not need to be known in order to recover it by reboot. The fact that rebooting can be done "blindly" is one of the reasons some practitioners frown upon its liberal use; however, we view the avoidance of lengthy diagnosis as one of the main strengths of reboot-based recovery and an important way to reduce MTTR. The ability to recover without diagnosis explains why rebooting is so widely used in large-scale systems [Bre01a, Pal02, Lev03]. As software becomes more complex and availability requirements more stringent, the willingness and ability to perform thorough diagnosis prior to recovery will decrease over time, making reboots more befitting. As hardware becomes faster, a reboot will become proportionally faster, and thus an increasingly compelling form of recovery.

1.2.4 Drawbacks of reboot

Rebooting has two principal drawbacks: loss of data and unpredictable recovery times.

While scrubbing corrupt data is beneficial, losing good data is not. For example, in a traditional, buffered UNIX filesystem, updates are kept in the volatile buffer cache for up to 30 seconds; should

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

an unexpected crash occur during that period, any data that had been written to the buffer cache, but not to disk, will be lost. This problem has been recognized in today's Internet services, which is why most now maintain all important data (including session state, such as shopping carts) in databases.

Another drawback of rebooting is that it can result in long, unpredictable recovery times. Data recovery procedures in systems handling large amounts of data can last many hours (e.g., filesystem checks, transaction log undo/redo). Modern systems recognize this problem; for example, the Oracle database server allows administrators to tune the rate of checkpointing, such that recovery time after crash does not exceed a configured upper limit [LGWJ01]. In the worst case, if there is a persistent fault (e.g., a failed disk or a misconfiguration), the system may never come back up and require instead some other form of recovery.

1.2.5 Benefits without drawbacks

We argued that improving availability by reducing time-to-recover may be more productive than increasing reliability; we also showed that rebooting provides an attractive approach to recovery in large-scale Internet systems. However, rebooting can be expensive; we mitigate this cost by reducing the scope of a reboot.

We propose the concept of *microreboot* – individual restart of fine-grained application component(s) – as a practical recovery mechanism that can achieve many of the benefits of whole-process restarts, but an order of magnitude faster and with an order of magnitude less lost work. When a system becomes unavailable, it is usually just a small subset of its components that are faulty, so rebooting the entire system would entail a lot of unnecessary recovery. Microrebooting offers the means for surgically recovering only what needs to be recovered. The universality of reboot-based recovery promises to make microreboots effective for a variety of components, and to remain effective despite code changes.

To get all the benefits of rebooting with as few drawbacks as possible, microrebooting requires strong modularity and isolation. Both can be exploited to confine failures to one, or a few components, and perform localized recovery on just the faulty ones. Tandem's process pairs [Bar81] took this observation from the domain of hardware to that of software, and advocated modularity and recovery at the level of processes. We push this notion further, to the sub-process level, and achieve the effects of a reboot/restart on a smaller scale.

Furthermore, we completely separate data recovery from process recovery. For microreboots to achieve maximum benefit, components must be "stateless" and keep all important application state

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

in specialized state stores, separate from the application logic; this way, application data does not get lost or corrupted during a microreboot. The separation of data management from application logic (or "data transformation") allows us to take data recovery out of the critical path of process recovery. Since application software fails far more often than data management software (e.g., databases), but takes considerably less time to recover, this separation makes sense: it reduces overall downtime by separating low-MTTF/low-MTTR components from the high-MTTF/high-MTTR ones.

All applications, legacy or newly-created, that abide by a small set of design principles, can be recovered using microreboots without any further changes to the application. With a cheap, universal recovery mechanism in place, it becomes possible to build systems that recover on their own, with humans intervening only when absolutely necessary. This further reduces MTTR, since recovery is now done in "machine time" rather than "human time." Rather than try to automate and speed up complex human-intensive recovery processes, we aim to equip systems with simpler, more effective recovery levers.

1.3 Thesis and Contributions

Our thesis is that, although large-scale software systems are unreliable, structuring them for fast, minimally-disruptive recovery is a practical, cost-effective way to make them highly available. The present dissertation makes three contributions toward proving this thesis:

Crash-only design

We define a set of principles for building large-scale programs that crash safely and recover fast; there is only one way to stop such software – by crashing it – and only one way to bring it up – by initiating recovery. Crash-only applications consist of stateless components that keep all important state in application-independent state stores.

Microreboot mechanism

We show that, with a small number of changes, runtime platforms can support a mechanism by which fine-grained components of crash-only applications are recovered through microreboot at the first indication of failure. We demonstrate that microrebooting can achieve many of the same benefits as a process restart in Java systems, while reducing unavailability by a factor of 50. We show that the combination of crash-only software and microrebooting is a better cost/dependability tradeoff compared to the approach of aiming for correct code and supporting diverse recovery mechanisms.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Failure management policies

We show that the fast, minimally-disruptive nature of microrebooting makes several failure management policies cost-effective, policies that would otherwise be prohibitively expensive in terms of incurred downtime. First, we show that failures can be avoided at low cost by preventively microrebooting components, thus rejuvenating applications with minimal downtime. Second, we show that microrebooting at the slightest hint of failure (without engaging in diagnosis) improves availability, even when failure detection is prone to false positives. Finally, we demonstrate that microreboot-based recovery can be hidden from end users via transparent request retries, improving availability without change in end-user-perceived service quality.

The microreboot approach is in keeping with a minimalist philosophy of system design, in which simpler recovery mechanisms are preferred to complex ones – by casting most failures as reboot-curable problems, we simplify recovery, making it more prompt and effective, while being less disruptive to end users.

1.4 Outline

The remainder of this dissertation consists of ten chapters and two appendixes. In chapters 2 and 3, we describe a selection of related work, as well as our early forays into the systematic use of (micro)rebooting to improve system availability, respectively. Based on this early experimentation, we formulated the crash-only design principles, which appear in chapter 4, and designed the microreboot recovery mechanism, described in chapter 5. We built a complete prototype of a crash-only system, in several variants; in chapter 6 we describe this prototype along with the framework used to evaluate the microreboot mechanism. In chapter 7 we demonstrate experimentally that substituting the microreboot mechanism for process restart in the recovery policy of a J2EE system reduces downtime by a factor of 50. We describe and evaluate the qualitative improvements introduced by microreboot-centric recovery policies in chapter 8. In chapter 9, we discuss the limitations of our work, along with challenges in applying it more broadly. Chapter 10 concludes the dissertation.

Appendix A describes a tool for inferring inter-component dependencies, used in determining groups of components that need to be microrebooted together. Appendix B provides a mapping from chapters in this dissertation to published papers.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 2

Related Work

The rebooting technique embodied in crash-only software and recursive microreboots has been around as long as computers themselves, and our work draws heavily upon decades of research and development history. This dissertation refines and systematizes a number of known techniques, turning them into a unitary, well-understood tool. In this chapter we describe some of the related work that provides the background for the research presented here. We describe the relationships between previous efforts and ours, whether the projects preceded ours or constitute extensions of previous systems, and also whether they solved a similar problem to ours but for a system with different constraints than those we address.

2.1 Preventing Software Failures

2.1.1 Better software

Many techniques have been advocated for improving software dependability, ranging from better software engineering [Bro95] and object oriented programming languages [DN66] to formal methods that predict/verify properties based on a mathematical model of the system [SM00]. Language-based methods, such as static analysis [Cou01], detect problems at the source-code level. Some programming languages prevent many programming errors by imposing restrictions, such as type safety [Wir88] or a constrained flow of control [MRA87], or by providing facilities like garbage collection [McC59].

This vast body of techniques have significantly improved software quality and, without them, it would be impossible to build today's software systems. At the same time, however, intricate

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

software leads to bugs that are hard to find, and much software today is written by developers with little training; hence the need for recovery-oriented techniques like the ones described in this dissertation. Our microreboot-based approach to recovery is complementary to all efforts aimed at improving the quality of software. As will be described at more length in section 9.2, the successful use of microreboots assumes that software has already undergone what would be considered today "diligent quality assurance."

2.1.2 Containing failure propagation

Fault containment techniques aim to confine faults, so they affect as little of a system as possible and allow for localized recovery. Good fault containment reduces the number of recovery attempts required to resolve a failure, resulting in faster recovery times. Drawing strong fault containment boundaries has long been considered good engineering and is found in many successful systems; for some, strong fault isolation is a fundamental principle [CRD$^+$95]. Techniques used for containment range from physical isolation for cluster nodes to hardware-assisted virtual memory and sophisticated software-based techniques. For example, the *taintperl* package [WS91] employs dynamic data flow analysis to quarantine data that may have been "contaminated" by user inputs and thus might contain malicious executable code – a serious security threat for Web sites using cgi-bin scripts. Applications already employing such techniques are more amenable to our localized recovery, but it is difficult to retrofit such approaches without significant changes to the applications.

Another set of containment technologies holds much more promise for microreboot-based recovery. Virtual machine monitors provide a powerful way to draw strong fault isolation boundaries between subsystems without having to change the application software. A virtual machine monitor [Gol74, BDGR97] is a layer of software between the hardware and operating system, which virtualizes all the hardware resources and exports a conventional hardware interface to the software above; this hardware interface defines a virtual machine (VM). Multiple VMs can coexist on the same real machine, allowing for multiple copies of an operating system to run simultaneously. In our research group we use virtual machines to isolate each publicly accessible network service (sshd, Web servers, etc.) from all the other services running on the same host: in each VM we run a copy of the OS and one single service. This way, a vulnerability in a Web server will not directly compromise any of the other services. Motivated by this type of VM use, isolation kernels [WSG02] provide a VM-like environment, but trade off completeness of the virtualized machine for significant gains in performance and scalability. While requiring slight modifications to the services, isolation

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

kernels provide a lightweight mechanism for building strong fault isolation boundaries. One isolation technique that we find particularly useful, given our choice of using a J2EE platform (see chapter 6), is the multi-tasking Java VM [CD01], which allows process-like isolation among applications running in the same JVM. Similar systems exist, such as JanosVM [THL01], which allows a single logical Java virtual machine to be split among multiple OS processes; and Luna [HvE02], which improves isolation among "tasks" running in a single Java VM.

The isolation of operating system services into separate components, for purposes that include containment, has been pioneered by microkernels [ABB+86]; more recent work has demonstrated that the performance overhead of achieving such isolation is negligible [HHL+97]. Containment of drivers [SABL04], identified as the buggiest part of operating system kernels today, has been employed successfully to save systems from crashing. All these projects enable the use of microreboots in operating system kernels.

Isolated processing components appeared also in pre-J2EE transaction processing monitors, where each type of system functionality (e.g., doing I/O with clients, writing to the transaction log) was a separate process communicating with the others using IPC or RPC. Session state was managed in memory by a dedicated component. Although the architecture did not scale very well, the "one component/one process" approach provided better isolation than monolithic architectures and would have been amenable to microrebooting.

Finally, the SEDA [WCB01] project proposed an architecture in which performance behaviors are isolated into separate stages, giving operators control over each stage individually. A slowdown in one stage will not affect another stage (but, of course, will impact overall throughput). SEDA also recognized the value of moving certain behaviors (admission control, load balancing, etc.) into the runtime system, such that all applications running on that platform would benefit. In SEDA's case, applications had to be written in an event-driven continuation-passing style; admission control and load balancing could then be done implicitly by the SEDA middleware. However, this requires recoding the application in a somewhat nonintuitive programming style. We embraced the concept of adding support into the platform, but aimed for minimal changes to applications themselves.

An interesting "counter"-containment approach was recently proposed under the name of failure-oblivious computing [RCD+04]. It is based on a C compiler that inserts checks that dynamically detect invalid memory accesses. Instead of terminating or throwing an exception, the generated code simply discards invalid writes and manufactures values to return for invalid reads, enabling the server to continue its normal execution path. Initial experimentation indicates that this technique is effective in improving availability for some applications. It is based on the same observation that

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

microreboots is based on: when the workload is broken into fine-grained, independent units, then recovery can be performed on a subset of the workload with little impact on the rest of the workload.

2.1.3 Software rejuvenation

Long-running software ages, i.e., the availability of system resources deteriorates, and data corruption and numerical error accumulate, eventually leading to failure. Software rejuvenation [HKKF95] terminates an application and restarts it at a clean internal state to prevent faults from accumulating and causing the application to fail. Rejuvenation has also found its way into Internet server installations based on clusters of hundreds of workstation nodes; many such sites use rolling reboots to clean out stale state and return nodes to known clean states, Inktomi being one example [Bre01a]. IBM's xSeries servers also employ rejuvenation for improved availability [Int01]. As will be seen in chapter 8, proactive microrebooting (microrejuvenation) is a valuable element of microreboot-centric recovery policies.

2.2 Detecting failures

Rapid detection is a critical ingredient of fast recovery and is therefore an integral part of any recovery-oriented approach to system dependability. A large fraction of recovery time, and therefore availability, is the time required to detect failures and localize them well enough to determine a recovery action [CAK+04]. A study [OGP03] found that earlier detection might have mitigated or avoided 65% of reported user-visible failures. By enabling "sloppier" fault detection, we make a number of detection and diagnosis solutions more useful. For example, statistical learning approaches [CKF+02], while prone to false positives, are useful for systems whose structure is not known a priori (see chapter 8).

Programmer-inserted assertions and periodic consistency checks – staples of defensive programming – are an excellent way to catch bugs. Database and telecommunications systems take this one step further and employ audit programs to maintain memory and data integrity. In the 4ESS telephone switch [Wil82], for example, so-called mutilation detection programs constantly run in the background to verify in-memory data structures. When a corrupt structure is found, its repair is attempted by a correction module; if the repair fails, the data structure is reinitialized. Such applications integrate well with the recursive microreboot framework presented here, as they can notify the recovery manager when detecting an unrecoverable fault, and allow the manager to decide what higher level recovery action to take.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Some of the most reliable computers in the world are guided by the principle of fast detection, and use dynamic recovery to mask failure from upper layers. For example, in IBM S/390 mainframes [SG99], computation is duplicated within each CPU and the results are compared before being committed to memory. A difference in results freezes execution, reverts the CPU to its state prior to the instruction, and retries the failed instruction. If the results now compare OK, the error is assumed to have been transient and execution continues; if they are different, the error is considered to be permanent, and the CPU is stopped and dynamically replaced with a spare CPU by reconfiguring data paths. Execution of the instruction stream resumes transparently to the operating system. In its memory system, the S/390 performs background scrubbing on main memory data to reduce the frequency of transient single-bit failures; faulty memory chips are dynamically replaced. While the S/390 is a reliable computer that hardly ever fails, in large software systems most downtime is not caused by hardware. For this reason, in our approach we perform fault detection at all levels in the system, thus being able to capture more failure scenarios than could be detected by the hardware alone. Moreover, the recursive recovery approach accepts that always choosing the right level in the system at which to recover is difficult, so it progressively tries higher and higher layers until the problem is eradicated.

BASE [RLC01] and BFT [CL99] try to detect and correct what would otherwise be silently-wrong answers, e.g. due to data corruption or a malicious adversary. Their work is complementary to ours and composes with it, though we note that the state corruption errors we encountered (see chapter 7.1) would have been difficult for these approaches to find.

Finally, some amount of knowledge about the application's semantics enables end-to-end failure detection. Infrastructure monitoring companies [NOC02, Res02, Com02, Int02, Hew02] actively supervise corporate databases, application servers, and Web servers by monitoring specific aspects (e.g., the alert log contents of a DBMS, the throughput level of a Web server). The infrastructure operator is notified when something has failed, is exhibiting fail-stutter behavior [ADAD01b], or when resource utilization is approaching application-specific critical levels and may warrant rejuvenation. A similar approach has been taken in gray-box systems [ADAD01a], where knowledge of the internal workings of an operating system is captured in information and control layers, which can then observe OS activity and infer facts about the OS state without using explicit interfaces. In our prototype we use score-based fault localization and tools like Pinpoint [KF05] to map end-to-end failures or performance degradation onto the specific components that are causing the failure and recover them with surgical precision.

ARMOR [WBS+98] provides some application-generic services, such as liveness monitoring

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

and reboot, to distributed applications. It provides checkpoint-based recovery to applications written to ARMOR's micro-checkpointing API, including ARMOR itself (i.e., the ARMOR middleware modules can recover using their own checkpoints). Compared to ARMOR, we are attempting to detect more classes of failures via the use of different types of plug-in failure monitors, and collecting in one place (the recovery manager) the policy decisions as to what should be rebooted to attempt recovery. In our work, we are interested in supporting any unmodified J2EE applications, which are not tied to a specific checkpointing API, but instead follow a set of design principles.

The monitors presented in this dissertation are strictly complementary to other failure detection techniques. The most common techniques for detecting failures in Internet services are low-level monitoring, such as heartbeats and pings, and periodic high-level, end-to-end application checks [MS00]. Heartbeats and pings have the advantage of being simple to implement and easy to maintain. However, they lack the ability to detect many application-level failures. Complex, end-to-end tests that make use of detailed application semantics are able to detect application-level failures, but, since they must be redeveloped for individual applications, they are expensive to build. In addition, they require significant maintenance to keep up-to-date with rapidly evolving applications.

2.3 Tolerating software failures

2.3.1 Redundancy

Virtually all recovery techniques rely on some form of redundancy, in the form of either functional, data, or time redundancy. In the case of functional redundancy, good processors can take over the functionality of failed processors, as in the case of Tandem process pairs [Bar81] or clusters [FGC+97]. Some forms of recovery use time redundancy and diversity of programming logic (e.g., recovery blocks [AK76], where the computation of an erroneous result triggers a retry using a different algorithm), but such techniques have had only limited appeal due to their cost of development and maintenance, as well as difficulty in ensuring true independence among the alternate program paths.

Redundancy and failover [MS00] are staples of Internet services and the most popular way to reduce downtime. The techniques presented here are complementary to that strategy, since failed nodes must eventually be recovered to restore system throughput, as well as close the "window of vulnerability" associated with operating under partial failure.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

The CNN.com meltdown on 9/11/01 [LeF01] is a good example of how slow node-level recovery time can lead to an entire service collapsing. Failover to a standby node is a powerful high availability technique, but it cannot be solely relied on, because node failure takes the cluster into a period of vulnerability in which further failures could cripple it. The CNN news site collapsed on 9/11/01 under the rapidly increasing load, because thrashing nodes could not recover quickly and good nodes could not reintegrate fast enough to take over from the thrashing ones [LeF01]. Our project's emphasis on reducing recovery time complements redundancy-based failover by reducing the system's window vulnerability to additional failures. Most Internet services run on very large clusters of computers (as an extreme example, Google uses tens of thousands of CPUs in 5 geographically distributed sites to serve google.com [Ach02]); at this scale, nodes become unavailable quite frequently, making rapid reintegration critical.

Finally, there are limits to the scalability of node redundancy, because of the tension between number of nodes and diversity. Having a large number of diverse nodes increases the system's robustness to failure, but at the same time makes it very difficult to administer and maintain, which makes the system prone to operator-induced failure. On the other hand, having a large number of mostly identical nodes makes management easier, but drastically reduces system robustness. For example, when an obscure bug in the software of a major content distribution network (CDN) manifested simultaneously on all nodes running that release, a large part of the CDN crashed, leading to a widespread outage. Rapid node recovery allows even widespread failure to be quickly eradicated.

Process pairs [Bar81] were an early mechanism that combined resource redundancy and state mirroring to allow failover to a hot standby, but because they were difficult for programmers to use, they have had limited impact outside of specialized high-end systems. Transactions [Gra81] have enjoyed much wider impact, and remain a key element of today's Internet applications, because they are easy for programmers to use and they export a clean abstraction for dealing with recovery. However, when combined with relational semantics, providing transactional guarantees requires substantial engineering in order to get both good steady-state performance and complete crash-safety and recovery (indeed, high-volume, high-performance database systems cost hundreds of thousands to millions of dollars to deploy and maintain).

Separating applications into stateless logic plus transactions simplifies recovery; we exploit this property by attempting application-generic recovery for the logic. We push this approach to its logical extreme, by specializing the state stores used for other kinds of Internet service state, including session state and persistent non-relational state such as user profiles.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

2.3.2 Saving state

Checkpointing [WHV+95, CR72, TJWR99] employs dynamic data redundancy to create a believed-good snapshot of a program's state and, in case of failure, return the program to that state. An important challenge in checkpoint-based recovery is ensuring that the checkpoint is taken before the state has been corrupted [WIH+02]. Another challenge is deciding whether to checkpoint transparently, in which case recovery rarely succeeds for generic applications [LCC00], or non-transparently, in which case source code modifications are required. In spite of these problems, checkpointing is a useful technique for making applications restartable, and was successfully utilized in [HK93], where application-specific checkpointing was combined with a watchdog daemon process to provide fault tolerance for long-running UNIX programs. ARMORs [KIBW99] provide a micro-checkpointing facility for application recovery, but applications must be (re)written to use it; limited protection is provided for legacy applications without their own checkpointing code. We believe that maintaining state in a suitable store (see chapter 4) obviates the need for checkpoints.

Past work [LCC00, CC00] has painted a grim picture regarding application-generic recovery, showing that general-purpose transparent recovery is unlikely to work. They formulated an approach to application-generic recovery (i.e. recovery without application-specific knowledge) based on checkpointing, and demonstrated that relatively few existing applications could be successfully recovered by this approach. They studied both Unix-style monolithic applications such as `vi` and large open-source Internet service components such as MySQL and Apache.

The authors' conclusion derives in part from the fact that a fully generic recovery system cannot make any assumptions about application structure, and therefore about what constitutes safe (correctness-preserving) generic recovery. We argue that for the specific class of applications we're targeting – interactive Web-connected services deployed in a traditional multi-tier configuration – we *can* make assumptions about their structural properties, and these assumptions make it possible to obtain application-generic benefits solely by modifying the middleware. Although we do exploit application-specific fault propagation information to guide recovery, the process for collecting this information is itself application-generic, automatic, and relatively fast to perform.

Part of the appeal of rebooting as a recovery technique is that it *discards* corrupted transient state that might itself be the cause of the failure or whose cleanup may be necessary in order for recovery to succeed. Therefore we expect that replacing recovery with rebooting – which is logically equivalent to restarting from a checkpoint that is the start state of the component – is more likely to work, assuming it is safe to try.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

2.3.3 Careful state updates

Log-based recovery techniques cost more than checkpoint-based recovery, but are considerably more powerful, because they allow the system to return to potentially any moment in time prior to the failure. Undo and redo logs [Gra78, RO91] allow the system to undergo a set of legal transformations that will take it from an inconsistent state, such as that induced by a bug or hardware failure, to a consistent one. Logs enable transactions [Gra81], which are the fundamental unit of recovery for applications that require ACID [Gra78] semantics. In a new twist on the undo approach, system-level undo [BP03] allows for an entire system's state to be rolled back, repaired, and then brought back to the present.

There is a long line of systems that have implemented the transaction concept [Gra81], and such systems are highly synergistic with reboot-based recovery. The transaction introduced the powerful and simple notion of recovering from failures while being oblivious to the reason for which that failure occurred. In microreboot-based recovery we exploit much of the same approach.

The Quicksilver system [HMC88] is particularly relevant to the work presented in this dissertation – it uses atomic transactions as a unified failure recovery mechanism for client-server distributed systems. In Quicksilver, transactions allow failure atomicity for related activities at a single server or at a number of independent servers. Rather than bundling transaction management into a dedicated language or recoverable object manager, Quicksilver exposes the basic commit protocol and log recovery primitives, allowing clients and servers to tailor their recovery techniques to their specific needs; servers can implement their own log recovery protocols rather than being required to use a system-defined protocol. These decisions allow servers to make their own choices to balance simplicity, efficiency, and recoverability. At the same time, however, Quicksilver places a significant burden on the developers of applications for this environment; in our work, we chose to abstract away from programmers as many of the intricacies of microrebooting as possible. We also do not advocate the universal use of transactional semantics, recognizing that in certain cases (such as session state objects), using log-based approaches to atomicity introduces unnecessary overhead.

Two interesting systems provide safety in the face of crashes and fast recovery, although they depart from transactional semantics: Sprite LFS [RO91] and WAFL [HLM94]. The log-structured file system (LFS) writes all modifications to disk sequentially in a log-like structure, thereby speeding up crash recovery. The log is the only structure on disk, although it does contains indexing information so that files can be read back from the log efficiently. Network Appliance introduced the file server appliance [HLM94], a dedicated server whose sole function is to provide NFS file service. The filesystem employed in this appliance is called WAFL (Write Anywhere File Layout) and

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

its primary focus is to implement "snapshots," which are read-only clones of the active file system. Snapshots eliminate the need for file system consistency checking after an unclean shutdown, which results in speedy reboot-based recovery. As described in chapter 4, this type of filesystems could be construed as "crash-only" state stores that play an important role in microrebootable systems.

2.3.4 Making recovery fast

The benefits of restarting quickly after failures have been recognized by many system designers, as they employed techniques ranging from the use of non-volatile memory (e.g., Sprite's recovery box [BS92] and derivatives of the Rio system [CNRA96, LC97]) to non-overwriting storage combined with clever metadata update techniques (e.g., the Postgres DBMS [Sto87], Network Appliance's filers [HLM94]). A common theme is the segregation and protection of state that needs to be persistent, while treating the rest as soft state. We see this approach reflected in recent work on soft-state/hard-state segregation in Internet services [FGC+97, GBHC00] and we adopt it as a basic tenet.

Baker [BS92] observed that emphasizing fast recovery over crash prevention has the potential to improve availability, and she described ways to build distributed file systems such that they recover quickly after crashes. In her design, a "recovery box" safeguards metadata in memory for recovery after a warm reboot. In our work, we provide components for a more general framework that both reduces the impact of a crash and speeds up recovery.

Other research systems have embraced the approach of reducing downtime by recovering at sub-system levels. For example, Nooks [SABL04] isolates drivers within lightweight protection domains inside the operating system kernel; when a driver fails, it can be restarted without affecting the rest of the kernel. Farsite [ABC+02], a peer-to-peer file system, has been recently restructured as a collection of crash-only components, that are recovered through rebooting. These systems provide examples of microrebootable systems and lend credibility to the belief that non-J2EE systems can be structured for effective microrebootability.

Autonomic computing [KC03] seeks to automate complex systems administration as much as possible, often by having a system automatically learn or infer its operating points and then applying automated management techniques based on closed-loop control or statistical modeling. Recent work in automatic inference of the behavior of complex applications relies on collecting fine-grained (component-level) observations and extracting interesting patterns from them [CZL+04, DMM04], whereas recent progress in applying automated management techniques [CNFC04, LCD+03] assume a predictable performance cost for triggering management mechanisms such as recovering or

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

activating a node. We combine similar techniques in the context of our three-tiered J2EE prototype system, building upon the observation that, when a machine can recover autonomously (see chapter 8), it can do so much faster than a human operator.

2.4 Chapter Summary

Prior work has generally assumed failure to be an exceptional occurrence, that required special handling. The result of this approach was an overemphasis on improving program reliability, while providing just enough recovery mechanism to handle such exceptional events. Unfortunately, recovery code is difficult to test and, when invoked, must run flawlessly, because the system must emerge from the failed state. In our work, we regard failure as a regular event, not an exceptional one; this perspective will become increasingly legitimate, as software failure rates will increase with program complexity.

In order to support the "failure as a regular event" paradigm, software must be written such that it can be recovered frequently, easily, and quickly. In this dissertation we identify the principles behind such a design, while minimizing the departure from common programming practice.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 3

Background

Prior to the work presented in this dissertation, we did a preliminary exploration of the concept of reboot-based recovery in two other systems. Here we briefly describe these earlier projects that set the stage for our main body of work: the Medusa execution environment (section 3.1) and the Mercury satellite ground station (section 3.2).

3.1 Medusa: An Execution Platform for UNIX Programs

Our first experience in using reboot-based recovery was in the context of Medusa [Can00]. This project's main contribution was to explore the possibility of building an execution platform that allows highly available services to be built from programs that (individually) do not exhibit such characteristics. The motivation for this work was our belief that perfect software is impossible to produce.

The Medusa system is a rudimentary self-managing, self-replicating platform that can execute arbitrary UNIX programs across a collection of UNIX machines. Medusa achieves increased end-to-end availability for a large class of applications by exploiting loose coupling between its components and quick reactive behavior in the face of failures.

3.1.1 Overview

The Medusa prototype is composed of a variable number of *segments*, which intercommunicate using UDP over IP multicast. A segment is the basic unit of process replication and has two core responsibilities: to ensure the well-being of other segments and to execute commands supplied by Medusa's clients. A segment runs on one host at any given point in time, but can migrate to

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

other machines when this becomes necessary; it provides a highly available execution container for generic UNIX programs.

Whenever a client wishes to execute a command on Medusa, it multicasts a special packet to the entire system. Individual segments inspect this packet and respond, if they are available to execute the command; a segment is ultimately elected to run the command. If the client does not hear back from Medusa, it resends its request. Should the executing command fail abnormally, either due to its own or its host segment's failure, it is immediately restarted, potentially on a different machine. The client can monitor the multicast communication between segments and determine, based on its content, the progress of the execution. A segment will never communicate directly with a client or another segment. In the event that the client who submitted an execution request dies while this request is being completed, the requested program continues running even after the client is gone.

Segments do not share any state, thus achieving maximal fault isolation. The collaborative nature of segments' activity is limited to monitoring, replicating, and restarting each other. Host machines and segments are logically independent, making Medusa's host configuration requirements very simple: (a) ensure that some version of the Secure Shell Daemon `sshd` is running, which is freely available and already common place on many UNIX machines, and (b) provide a login account for Medusa.

This independence further enables simple upgrades through injection of newer Medusa segments into the system without having to upgrade individual machines.

Medusa could be regarded as a two-layered watchdog. Each segment is watched by its peer segments, that ensure the segment continues running and doing its work regardless of failures. At the same time, the segment itself is a watchdog for the application it is executing, ensuring the application "stays up" even in the face of failures.

Each individual segment has two orthogonal threads of control: a heartbeat generator and the segment's state machine. Heartbeats provide the basis of Medusa's health maintenance mechanism, while the segment state machine is responsible for processing other segments' heartbeats, replicating and restarting, as well as executing commands. Not hearing from a segment for some amount of time is taken as an indication that the segment has died or been subjected to a network partition. In the current prototype, the expiration time interval is a configurable multiple of the inter-heartbeat sending intervals, but ideally would be a function of the heartbeat arrival rate from the monitored segment. When a segment is declared dead, it must be regenerated by another segment through replication.

Medusa can provide high availability for applications that have the following characteristics:

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

- Application is restartable, i.e., running it again from the beginning is a recoverable form of operation. In most cases, such applications can be built with reasonably low effort by using a transactional substrate.

- Application's operations must be globally idempotent; in the event that Medusa restarts the application, its actions may repeat prior actions and so the end result must not be affected by repeated executions. Most Internet services have this property.

- Sequential commands cannot require Medusa to maintain state on their behalf. For example, the current prototype would not support running a command such as `cat /etc/passwd` and then having Medusa feed this execution's output into a new command, such as `grep /u1`. If this sequence is desired, it must be requested as a unitary command (e.g., "`cat passwd | grep /u1`").

Medusa emphasizes manageability. New segments can smoothly join the system, which means it can scale up (or down) seamlessly. Medusa currently requires virtually no configuration, except for a handful of command line arguments; this is considered a virtue because it reduces the risk of operator errors. We used Medusa as a way to explore the paradigm of network-wide software upgrades performed through simple injection of higher versioned segments, which gradually replace old Medusa segments. The hope was that, one day, we will not view software as packaged entities anymore, rather as populations that get released into computing communities.

3.1.2 Lessons

The Medusa project illustrated the viability of building an execution platform for services with high availability requirements. It allowed generic programs to exhibit increased end-to-end robustness by running on a self-managing, self-replicating base. The key architectural decisions included partitioning Medusa into loosely coupled segments, embedding highly reactive, adaptive behaviors, and using simple, uniform multicast-based communication. Medusa provided a starting point and experience for developing a dependability toolkit for the average programmer, consisting of a platform and a set of tools for building highly dependable software infrastructures from COTS components. Some of the underlying design choices were based on principles derived from successful biological systems: diversification, adaptation, and large-scale replication. The envisioned beneficiaries of this toolkit were developers of mission/business-critical Internet services and pervasive computing infrastructures.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

3.2 Mercury: Control Software for a Satellite Ground Station

Our second system exploring reboot-based recovery was the control software for the Mercury satellite ground station [CCF+02]. We collaborated with the Space Systems Development Lab (SSDL) on the design and deployment of space communications infrastructure to make collection of satellite-gathered science data less expensive and more reliable. One necessary element of satellite operations is a ground station, a fixed installation that includes tracking antennas, radio communication equipment, orbit prediction calculators, and other control software. When a satellite appears in the patch of sky whose angle is subtended by the antenna, the ground station collects telemetry and data from the satellite.

When we approached the design and deployment of the Mercury software, we wanted to improve ground station availability – the control software had not been originally designed with high availability in mind and was not written by professional programmers, yet it was used for several active, in-orbit satellites. Our first step in improving the availability of Mercury was to apply recursive microrebooting to "cure" transient failures by restarting suitably chosen subsystems, such that overall mean-time-to-recover (MTTR) was minimized.

We had two main goals in applying recursive microreboots to Mercury. The first was to partially remove the human from the loop in ground station control by automating recovery from common transient failures we had observed and knew to be curable through full restarts or microreboots. In particular, although all such failures are curable through a brute force reboot of the entire system, we sought a strategy with lower MTTR. The second goal was to identify design guidelines and lessons for the systematic future application of microreboot-based recovery to other systems.

3.2.1 Overview

The general software architecture is shown in figure 3.1: fedrcom is a bidirectional proxy between XML command messages and low-level radio commands; ses (satellite estimator) calculates satellite position, radio frequencies, and antenna pointing angles; str (satellite tracker) points antennas to track a satellite during a pass; rtu (radio tuner) tunes the radios during a satellite pass; mbus passes XML-based high-level command messages between software components.

The ground station components are safe to reboot, since they do not maintain persistent state; they use only the state explicitly encapsulated by received messages from mbus. Hard state exists in Mercury, but is read-only during a satellite pass and is modified off-line by ground station users. In addition, the set of Mercury failures that can be successfully cured by reboot is large, and in fact this

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

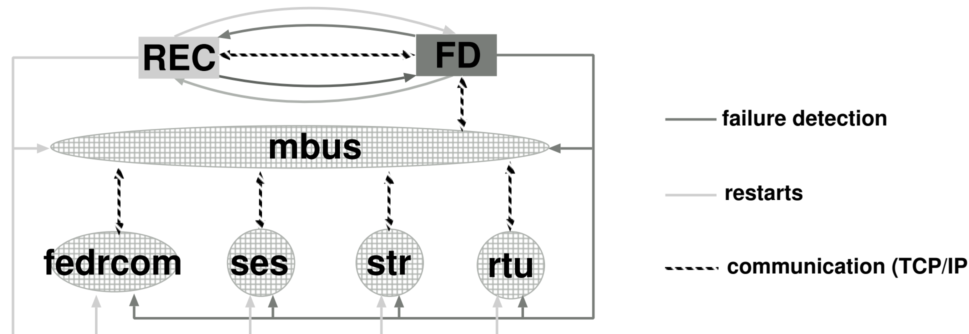

Figure 3.1: Mercury software architecture

is how human operators recovered from most Mercury failures before we implemented automated recovery.

Mercury is a soft-state system, in that any writable state is constantly refreshed by messages, and state which is not refreshed eventually expires. Soft state and announce/listen protocols have been extensively used at the network level before [ZDE^{+}93, DEF^{+}96] as well as the application level [FJLM95]. Announce/listen makes the default assumption that a component is unavailable unless it says otherwise; soft state can provide information that will carry a system through a transient failure of the authoritative data source for that state. The use of announce/listen with soft state allows restarts and "cold starts" to be treated as one and the same, using the same code path; this is an important tenet of crash-only software (chapter 4). Moreover, complex recovery code is no longer required, thus reducing the potential for latent bugs and speeding up recovery. In soft-state systems, reboots are guaranteed to bring the system back to its start state; by definition, no data corruption is possible.

Mercury's failure detection architecture was based on the addition of two new independent processes: a failure monitor (FD) and a recovery manager (REC). FD continuously performs liveness pings on Mercury components, with a period of 1 second, determined from operational experience to minimize detection time without overloading mbus. When FD detects a failure, it tells REC which component(s) appear to have failed, and continues its failure detection. For improved isolation, FD and REC communicate over a separate dedicated TCP connection, not over mbus; mbus itself is monitored as well.

Given the above strategy, two situations can arise, which we handle with special case code. First, FD may fail, so we wrote REC to issue liveness pings to FD and detect its failure, after which it can restart FD. Second, REC may go down, in which case FD detects the failure and restarts REC,

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

although the generalized procedural knowledge for how to choose the modules to microreboot and initiate recovery is only in REC.

Two salient properties of Mercury distinguish it from larger-scale Internet applications. First, this is a static system that does not need to evolve online; it can be upgraded and reconfigured inbetween satellite passes. Second, there are no circular functional dependencies between components, and in particular, its fault propagation and recovery maps are very simple and have a tree structure. In subsequent work, we operated on systems with more general dependency structures (see chapter 6 and appendix A).

3.2.2 Lessons

An interesting principle we found during this work was that, if one adopts a recovery strategy based on microreboots, component boundaries should take into account MTTR and MTTF, rather than be based solely on "traditional" modularity considerations such as state sharing. Redrawing the boundaries of software components based on their MTTF and MTTR helped us minimize overall system MTTR by enabling the tuning of which components are rebooted together. In contrast, most current system and software engineering approaches establish software component boundaries based solely on considerations such as performance overhead and amount of communication between components or amount and granularity of state sharing.

For example, there was one component (fedrcom) that connected to a serial port at startup and negotiated communication parameters with the radio device; thereafter, it translated commands received from the other components to radio commands. Due to the hardware negotiation, restarting fedrcom takes a long time; at the same time, due to instability in the command translator, it crashes often. Hence fedrcom has high MTTR and low MTTF. We split fedrcom into a pbcom component, which maps a serial port to a TCP socket, and fedr, the front end driver-radio that connects to pbcom over TCP. pbcom is simple and very stable, but takes a long time to recover (over 21 seconds); fedr is buggy and unstable, but recovers fast (under 6 seconds). After restructuring the code, our system recovery time improved by approximately a factor of 4.

An already well-known principle was brought to light by experimentation in Mercury, namely that unplanned downtime is generally more expensive than planned downtime, and downtime under a heavy or critical workload is more expensive than downtime under a light or non-critical workload. In Mercury, downtime during satellite passes (typically about 4 per day per satellite, lasting about 15 minutes each) is very expensive because we may lose some science data and telemetry from the satellite. Additionally, if the failure involves the tracking subsystem and the recovery time is too

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

long, the communication link will break and the entire session will be lost. A large MTTF does not guarantee a failure-free pass, but a short MTTR can provide high assurance that we will not lose the whole pass as a result of a failure. From among systems with the same level of availability, those that have lower MTTR are often preferable.

3.3 Chapter Summary

The two projects described here represent early forays into reboot-based recovery. Experimentation with the Medusa execution platform led us to the belief that a significant number of UNIX applications are restartable and that a system based on such applications can achieve reboot-based high availability in spite of the individual pieces being unreliable.

The Mercury ground station further indicated that the benefit of faster recovery has discontinuities (e.g., the threshold imposed by the duration of a satellite pass) that can be fruitfully exploited; this observation was developed further in [FP02]. Mercury was the first system in which we formulated the benefit of a good MTTR/MTTF balance in component design.

In the following chapter we will take a systematic approach to reboot-based recovery and explore the design principles that underlie reboot-friendly systems.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 4

Crash-Only Software

Despite decades of research and practice in software engineering, latent and pseudo-nondeterministic bugs abound in complex software systems; as complexity increases, they multiply further, making it difficult to achieve high availability. Transient faults account for a large fraction of failures in today's Internet systems and production software in general [MG95, AIS+01]; even mainframe-class operating systems are not immune to such transients [SC91]. Running out of memory or file descriptors, bug-triggering load spikes, deadlocks, performance degradation due to unexplained interactions between subsystems, etc. are just a few examples of what Internet service operators face on a regular basis [Lev03, Cho97]. It is common for such bugs to cause a system to crash, deadlock, spin in an infinite loop, livelock, or to develop such severe state corruption (memory leaks, dangling pointers, damaged heap) that the only high-confidence way of continuing is to restart the process or reboot the system. The Gartner Group [Sco99] estimates that 40% of unplanned downtime in business environments is due to application failures; 20% is due to hardware faults, of which 80% are transient [Cho97, MMS+00], hence resolvable through reboot.

When failure strikes business-critical software systems, operators cannot always afford to run real-time diagnosis. Instead, they focus on bringing the system back up by any means possible, and then do the diagnosis later. Our challenge is to find a simple, yet practical and effective way to build large, complex systems that are amenable to failure management through reboot, accepting the fact that bugs in application software will abound for as long as humans write the software. In this chapter, we first analyze in depth the benefits and drawbacks of recovering from failures via rebooting (Section 4.1), expanding on the analysis we presented in chapter 1. Afterward we argue for a design that assumes crash-rebooting to be a normal, frequent occurrence (Section 4.2), which takes the form of reactive restarts of failed components ("revival") as well as prophylactic restarts of

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

functioning components ("rejuvenation") to prevent state degradation. We then present a canon of design principles for reboot-friendly systems (Section 4.3), which collectively form the principles of "crash-only software." Finally, we discuss the benefits, drawbacks, and challenges related to designing crash-only systems (Section 4.4).

4.1 Reboot-based Recovery

The rebooting "technique" has been around as long as computers themselves, and remains a fact of life for substantially all nontrivial systems today. Although rebooting a system or restarting a process is often only a crude "sledgehammer" for maintaining system availability, its use is motivated by several properties:

4.1.1 Rebooting works around Heisenbugs

Most software bugs in production quality software are Heisenbugs [MD99, Cho97, Gra86, Ada84]. They are difficult to reproduce, or depend on the timing of external events, and often there is no other way to work around them but by rebooting. Even if the source of such bugs can be tracked down, it may be more cost-effective to simply live with them, as long as they occur sufficiently infrequently and rebooting allows the system to work within acceptable parameters.

Deadlock resolution in commercial database systems is a good example of living with application-level failures. It is typically implemented by killing and restarting a deadlocked thread in hopes of avoiding a repeat deadlock [Gra78]. The premise is that debugging deadlock-causing bugs in applications that use a database should not be in the critical path of recovering those applications, hence the push of this functionality down into the database layer.

Furthermore, the time to find and deploy a permanent fix can sometimes be intolerably long. For example, the Patriot missile defense system, used during the Gulf War, had a bug in its control software that could be circumvented only by rebooting every 8 hours. Delays in sending a fix or information about the reboot workaround to the field resulted in the system failing to intercept an incoming Scud missile, which hit the US Army barracks in Dahran (Saudi Arabia), leading to 28 dead and 98 wounded soldiers [Off92].

4.1.2 Rebooting returns system to a known state

Restarting a failed process or system reclaims stale resources and cleans up corrupt state, returning the system to a known, well-tested state, albeit with possible loss of data integrity. Corrupt or stale

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

state, such as a mangled heap, can lead to some of the nastiest bugs, causing extensive periods of downtime. Even if a buggy process cannot be trusted to clean up its own resources, entities with hierarchically higher supervisory roles (e.g., the operating system) can cleanly reclaim any resources used by the process and restart it.

In the Inktomi search engine, cluster nodes are periodically rebooted several times a day, in a rolling fashion, in order to bring the Web servers back to their initial, clean state [Bre00]. This is affordable because the cluster has n instances of the Web server for a population of u users, with each server being able to handle in excess of u/n users. A node reboot or transient node failure result solely in a decreased amount of search answers per query, while keeping overall query throughput constant.

4.1.3 Rebooting is simple and unequivocal

Unlike most recovery techniques, reboot-based recovery does not require the cooperation of the entity being recovered, but rather only that of its hierarchically-superior entity (e.g., a process can be restarted regardless of what that process is doing, as long as the kernel cooperates).

In addition to being unequivocal, rebooting is also straightforward to use and/or script; in an ideal world, all recovery from failures would take the form of some type of reboot. As we will see later, this makes reboot-based recovery easy to automate. Case in point: at major Internet portals, it is not uncommon for newly hired engineers to write and deploy production code after little more than one week on the job. Simplicity is stressed above all else, and code is often written under the explicit assumption that it will necessarily be killed and restarted frequently. This affords programmers such luxuries as never calling `free()` in their C code, thereby avoiding an entire class of pernicious bugs [Pal02].

4.1.4 Rebooting is effective against poorly understood failures

As described earlier, in practice reboot-based recovery "cures" many application-level failures [Mit04, Vor03, Lev03, Pal02]. This observation speaks not only to the effectiveness of reboots, but also to the fact that they can be employed against problems with unknown root causes. As software complexity increases, the number of unexplained behaviors increases as well, resulting in many failures for which the only recourse is reboot, hence the popularity of this technique. As we will show later, reboot is one of the few application-generic recovery techniques available today.

NASA's Mars Pathfinder illustrates the value of such recovery – shortly after landing on Mars,

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

the spacecraft identified that one of its processes failed to complete execution on time, so the control software decided to restart all the hardware and software [Ree98]. Despite the fact that the software was imperfect — it was later found that the hang had been caused by a hard-to-reproduce priority-inversion deadlock — the watchdog timers and restartable control system brought the system back into normal operation. It wasn't perfect software that saved the mission, rather the restart-oriented design of VxWorks, the operating system running on the spacecraft.

Rebooting is not usually considered a graceful way to keep a system running – most systems are not designed to tolerate unexpected crash-restarts, hence experiencing extensive and costly downtime when rebooted, as well as potential data loss. This occurs most frequently when the software lacks clean separation between data recovery and process recovery.

4.1.5 Rebooting can result in data loss/corruption

Reboot-based recovery in systems that are not crash-safe is dangerous. This is particularly acute in software with high performance requirements – corresponding tradeoffs often make programs more fragile. For example, why is it not safe to shut down a workstation by just flipping its power switch? Often the reason is performance tradeoffs made in the filesystem and other parts of the software. To avoid synchronous disk writes, many operating systems cache metadata updates in memory, opening a window of vulnerability during which allegedly-persistent data is stored only in volatile memory. An unexpected crash and reboot restarts the system's processes, but buffered data is lost, leaving the file system in an inconsistent state.

4.1.6 Rebooting can induce lengthy recovery

The filesystem example described above will usually require a lengthy `fsck` or `chkdsk` to repair, an inconvenience that could have been avoided by shutting down cleanly. The designers of such operating systems traded data safety and recovery performance for improved steady state performance.

In software systems not designed for restartability, the transient failure of one or more components often ends up being treated as a permanent failure. Depending on the system's design, recovering from a crash-induced failure can take very long if it requires manual intervention. NFS [SGK+85] exhibits a flavor of this problem in its implementation of locking: a crash in the lock subsystem can result in an inconsistent lock state between a client and the server, which sometimes requires manual

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

intervention by an administrator to repair. The result is that many applications requiring file locks test whether they are running on top of NFS and, if so, perform their own locking using the local filesystem, thereby defeating the NFS lock daemon's purpose.

In the rest of this chapter we will argue for a design in which we *decouple data recovery from process recovery*, i.e., separate the definition and preservation of critical state from the code that transforms that state. As will be shown later, such a design makes reboot-based recovery of processes safe and fast. This separation is the very basis of crash-only design, which aims to get the benefits of reboot-based recovery while mitigating its drawbacks

4.2 Why Crash-Only Design ?

In addition to arguing for a reboot-friendly design, we advocate software structure in which crashing is the only way of shutting down – this is the *crash-only design*.

Occam's Razor and the Restart Potpourri

As we have seen by now, rebooting is not optional in any system of reasonable size. There are many reasons to restart software, and many ways to do it, with most non-embedded systems having a variety of ways to stop; for example, an operating system can shut down cleanly, panic, hang, crash, lose power, etc.

When shutting down programs cleanly, the resulting unavailability consists of the time to shut down and the time to come back up; when crash-rebooting, unavailability consists only of the time to recover. Ironically, shutting down and reinitializing can sometimes take longer than recovering from a crash. Table 4.1 illustrates an informal comparison of reboot times (no critical data was lost in either of the experiments).

System	Clean reboot	Crash reboot
RedHat 8 Linux (with Ext3 filesystem)	104 sec	75 sec
JBoss 3.0 application server	47 sec	39 sec
Windows XP	61 sec	48 sec

Table 4.1: Duration of clean vs. crash reboots.

It is impractical to build a system that is guaranteed to never crash, even in the case of carrier class phone switches or high-end mainframe systems. Since crashes are unavoidable, software must be at least as well prepared for a crash as it is for a clean shutdown. But then – in the spirit

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

of Occam's Razor – if software is crash-safe, why support additional, non-crash mechanisms for shutting down? A frequent reason is the desire for higher performance.

We described earlier the performance tradeoff in some UNIX filesystems by which metadata updates are maintained in a write-back cache – this increases filesystem performance but also leaves the filesystem in an inconsistent state after a crash. Not only do such performance tradeoffs impact robustness, but they also lead to complexity by introducing multiple ways to manipulate state, more code, and more APIs. The code becomes harder to maintain and offers the potential for more bugs – a fine tradeoff, if the goal is to build fast systems, but a bad idea if the goal is to build highly available systems. If the cost of such performance enhancements is dependability, perhaps it's time to reevaluate our design strategy.

We define a crash-only system as one that obeys two equations: *stop=crash* and *start=recover*. The only way to stop the system is by crashing it; as a result, the only way to start the system is by initiating recovery.

Mature engineering disciplines rely on macroscopic *descriptive* physical laws to build and understand the behavior of physical systems. These sets of laws, such as Newtonian mechanics, capture in simple form an observed physical invariant. Software, however, is an abstraction with no physical embodiment, so it obeys no physical laws. Computer scientists have tried to use *prescriptive* rules, such as formal models and invariant proofs, to reason about software. These rules, however, are often formulated relative to an abstract model of the software that does not completely describe the behavior of the running system (which includes hardware, an operating system, runtime libraries, etc.). As a result, the prescriptive models do not provide a complete description of how the implementation behaves in practice, because many physically possible states of the complete system do not correspond to any state in the abstract model.

With the crash-only property, we are trying to impose, from outside the software system, macroscopic behavior that coerces the system into a simpler, more predictable universe with fewer states and simpler invariants. Each crash-only component has a single idempotent "power-off switch" and a single idempotent "power-on switch"; the switches for larger systems are built by wiring together their subsystems' switches. A component's power-off switch implementation is entirely external to the component, thus not invoking any of the component's code and not relying on correct internal behavior of the component. Examples of such switches include `kill -9` sent to a UNIX process, or turning off the physical, or virtual, machine that is running some software inside it.

Keeping the power-off switch mechanism external to components makes it a high confidence "component crasher." Consequently, every component in the system must be prepared to suddenly

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

be deactivated. Power-off and power-on switches provide a very small repertoire of high-confidence, simple behaviors, leading to a small state space. Of course, the "virtual shutdown" of a virtual machine, even if invoked with `kill -9`, has a larger state space than the physical power switch on the workstation, but it is still simpler than the state space of a typical program hosted in the VM, and it does not vary for different hosted programs. Indeed, the fact that virtual machines are relatively small and simple compared to the programs they host has been successfully invoked as an argument for using VMs for inter-application isolation [WSG02].

Recovery code deals with exceptional situations, and must run flawlessly. Unfortunately, exceptional situations are difficult to handle, occur seldom, and are not trivial to simulate during development; this often leads to unreliable recovery code. In crash-only systems, however, recovery code is exercised every time the system starts up, which should ultimately improve its reliability. This is particularly relevant given that the rate at which we reduce the number of bugs per thousand lines of code lags behind the rate at which the number of lines of code per system increases (see Section 1.1.2), with the net result being that the number of bugs in an evolving system increases with time [CYC+01]. More bugs mean more failures, and systems that fail more often will need to recover more often.

Many of the benefits resulting from a crash-only design have been previously obtained in the data storage/retrieval world with the introduction of transactions. Our approach aims for a similar effect on the failure properties of Internet systems – crash-only design is in many ways a generalization of the transaction model. It is important to note that Internet applications do not have to use transactions in order to be crash-only; in fact, ACID semantics are sometimes overkill. For example, session data accumulates information at the server over a series of user service requests, for use in subsequent operations. It is mostly single-reader/single-writer, thus not requiring ordering and concurrency control. The richness of a query language like SQL is unnecessary, and session state usually does not persist beyond a few minutes. These observations are leveraged by SSM [LF03], a crash-only hashtable-like session state store.

A crash-only system makes it affordable to transform every detected failure into component-level crashes; this leads to a simple fault model, and components only need to know how to recover from one type of failure. For example, [NBMN02] forced all unknown faults into node crashes, allowing the authors to improve the availability of a clustered Web server. Existing literature often assumes unrealistic fault models (e.g., that failures occur according to well-behaved tractable distributions); a crash-only design enables aggressive enforcement of such desirable fault models, thus increasing the impact of prior work. If we state invariants about the system's failure behavior and

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

make such behavior predictable, we are effectively coercing reality into a small universe governed by well-understood laws.

4.3 Principles of Crash-Only Design

In this section we describe a set of properties that we deem sufficient for a system to be crash-only both at the system level and at the component level. If components are crash-only, restarting these components becomes safe and fast; in chapter 7, we will show such component restarts can achieving many of the same benefits as whole-system restarts, but an order of magnitude faster and with an order of magnitude less lost work. Component-level restarts are called microreboots, a detailed description of which appears in chapter 5.

Retrofitting systems that are recoverable by reboot is generally a difficult task. Researchers have shown that the use of checkpoint-based solutions in an attempt to achieve application-generic recovery is challenging [LCC00]. It therefore makes sense to architect systems from the ground up, if the end result is "universal" reboot-based recovery. Starting from the hypothesis that applications are the predominant sources of downtime, rather than the data management servers, we describe here a few architectural guidelines that aim to separate process recovery from data recovery. In this vein, we have 3 goals:

- Strong boundaries: There must be a clear boundary around what is being rebooted, i.e., it should be possible to indicate unambiguously what state will be lost, what resources released, what loci of control returned to their start state, etc. For example, in the case of a process, the boundary is typically the process's heap and any kernel data structures or resources being maintained on the process's behalf.

- Loose coupling: if the entity being rebooted is part of a distributed system, other entities that communicate with it must be able to tolerate the reboot event as normal, not exceptional. For example, in a distributed system, calls to an RPC server that has failed and is in the process of recovering could be stalled or temporarily rerouted to a failover RPC server.

- State and consistency preservation: To avoid data loss, we must ensure that all state visible outside the component is either soft/discardable from the point of view of other components, or it is committed to a separate persistent state store which has its own recovery procedures in case of failure. For example, UDP multicast tree information is discardable soft state whose reconstruction is explicitly part of the corresponding routing protocol.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

These conditions do not guarantee that rebooting will successfully recover every observed failure, only that a reboot will not result in a change in application semantics (e.g., as caused by data or consistency loss).

To make components crash-only, we require that all critical state be kept in dedicated, crash-only state stores. To make a system of interconnected components crash-only, it must be designed so that components can tolerate the crashes and temporary unavailability of their peers. This requires strong modularity with relatively impermeable component boundaries, timeout-based communication and lease-based resource allocation, as well as self-describing requests that carry a time-to-live and information on whether they are idempotent. While recognizing that some of these choices sacrifice performance, we strongly believe the time has come for robustness to claim its status as a first-class citizen.

4.3.1 Fine-grained isolation

Component-level reboot time is determined by how long it takes for the underlying platform to restart a target component and for this component to reinitialize. A crash-only, microrebootable application therefore aims for components that restart as fast as possible, given application-specific constraints (such as functionality, permissions, etc.).

Components are defined by externally-enforced boundaries, that provide strong fault containment. The desired isolation can be achieved with virtual machines, isolation kernels [WSG02], task-based intra-JVM isolation [SDLS02, CD01], OS processes, etc. Web hosting service providers often use multiple virtual machines on one physical machine to offer their clients individual Web servers they can administer at will, without affecting other customers. The boundaries between components delineate distinct, individually recoverable stages in the processing of requests.

While partitioning a system into components is an inherently system-specific task, developers can benefit from existing component-oriented programming frameworks. Architectures that provide (and/or enforce) a component structure "out of the box," such as Microsoft's .NET and Sun's Java 2 Enterprise Edition (J2EE), provide an excellent platform for studying microreboots, and we will do so in chapters 6 and 7.

Aiming for fine-grained components is not a new goal; e.g., microkernels [ABB+86] have advocated the separation of operating system services into separate processes, albeit for different reasons than microrecovery. Yet, these efforts have seldom succeeded, primarily because the dependability benefits were not compelling when set against the performance overheads. We feel that now the time is ripe because: (a) complexity is forcing developers into modularized architectures for

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

reasons that go beyond recovery (maintainability, portability, testing, etc.); and (b) hardware has become so fast that performance overheads due to such structuring become inconsequential in increasingly more systems, with the benefits of the architecture far outweighing the performance drawbacks. Many large-scale infrastructures, often found in Internet services, have found ways to improve performance via horizontal scaling in clusters, in a way that does not compromise the modular architecture.

4.3.2 State segregation

We define *critical state* in an interactive system to be state that can only be recreated by replaying the end user interaction with the system. All other state (i.e., that can be recreated by the system on its own), is discardable upon reboot. For recovery to be correct, we must prevent microreboots from inducing corruption or inconsistency in the critical state. The inventors of transactional databases recognized that segregating recovery of persistent data from application logic can improve the recoverability of both the application and the data that must persist across failures. We take this idea further and require that microrebootable applications keep *all* critical state in dedicated state stores located outside the application, safeguarded behind strongly-enforced high-level APIs. Specialized state stores (e.g., relational and object-oriented databases, file system appliances, distributed data structures [GBHC00], non-transactional hashtables [HF03], session state stores [LF03], etc.) are better suited to manage state than application code.

Storing all critical state in dedicated state stores takes the (potentially lengthy) task of data recovery out of the critical path of application recovery, because the application can be restarted and recovered without triggering data recovery in the state store. In some sense, applications become stateless clients of the state stores, allowing applications to have simpler and faster recovery routines. A popular example of such separation can be found in three-tiered Internet architectures, where the middle tier is largely stateless and relies on back-end databases to store data.

Aside from enabling safe microreboots, the complete separation of data recovery from application recovery generally improves system robustness, because it shifts the burden of data management from the often-inexperienced application writers to the specialists who develop state stores. While the number of applications is vast and their code quality varies, database systems and session state stores are few and their code is consistently more robust. In the face of demands for ever-increasing feature sets, application recovery code that is both bug-free and efficient will likely be increasingly elusive, so data/process separation could improve dependability by making process recovery simpler. The benefits of this separation can often outweigh the potential performance

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

overhead.

Interfaces for crash-only state stores

Crash-only programs work best with crash-only state stores, because if the entire system is crash-only, a single type of recovery mechanism and policy can be used throughout. We therefore argue that dedicated state stores should be crash-only; this requirement does not simply push the problem down one level, but rather separates the part of the system that breaks often (applications) from the one that doesn't (data managers). Many commercial off-the-shelf state stores available today are crash-safe (i.e., they can be crashed without loss of data), such as databases and the various network-attached storage devices, but most of them recover slowly, making them poor crash-only state stores. The same products, however, offer tuning knobs that permit the administrator to trade performance for improved recovery time, such as taking checkpoints more often in the Oracle DBMS [LGWJ01].

The abstractions and guarantees provided by state stores must be congruent with application requirements. This means that the state abstraction exported by the state store should not be too powerful (e.g., offering a SQL interface with ACID semantics for storing and retrieving simple key-value tuples) and not too weak. A state abstraction that is too weak will require client components to do too much of their own state management, such as implementing a customer record abstraction over an offered file system interface. Good state abstractions allow applications to operate at their "natural" semantic level. Offering the weakest state guarantees that satisfy the application allows us to exploit application semantics and build simpler, faster, more reliable state stores.

For example, Berkeley DB [OBS99] is a storage system supporting B+tree, hash, and record abstractions. It can be accessed through four different interfaces, ranging from no concurrency control/no transactions/no disaster recovery to a multi-user, transactional API with logging, fine-grained locking, and support for data replication. Applications can use the abstraction that is right for their purposes and the underlying state store optimizes its operation to fit those requirements at the highest level of performance it can provide. Workload characteristics can also be leveraged by state stores; e.g., expecting a read-mostly workload allows a state store to utilize write-through caching, which can significantly improve recovery time and performance.

How many different state stores?

Enterprise and Internet applications are standardizing on a small number of state store types: transactional persistent state, single-reader/single-writer persistent state (e.g., user profiles, that almost

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

never see concurrent access), expendable persistent state (server-side information that could be sacrificed for the sake of correctness or performance, such as clickstream data and access logs), session state (e.g., the result set of a previous search, subject to refinement), soft state (state that can be reconstructed at any time based on other data sources), and volatile state. While differentiated mostly by guaranteed lifetime, the requirements for these categories of state lead to qualitatively different implementations.

An interesting example is offered by object-oriented databases (OODB), which provide persistent storage for programming language objects and offer a high level of congruence between the data model for the application and the data model of the database. Despite these advantages, OODBs have not shaken the market stronghold of the proven commercial relational databases (RDBMS). Instead, RDBMSs are increasingly offering object-oriented features, and at the same time allowing users to relax ACID requirements (e.g., by adjusting the inter-transaction isolation levels).

4.3.3 Inter-component decoupling

In an interactive system, subsystems that are recovered by crash-rebooting become temporarily unavailable to serve requests. For a crash-only system to gracefully tolerate such behavior, components need to be decoupled from each other. Strong decoupling between components is necessary for independent recovery, because components need to be smoothly re-integrated into the running system.

State segregation by itself introduces a certain level of decoupling, because separating components' process recovery from their data recovery implies that most component-level recovery can proceed independently of other components. Components in a crash-only system have well-defined, well-enforced boundaries; direct references, such as pointers, do not span these boundaries. If cross-component references are needed, they are stored outside the components, either in the application platform (operating system, application server, etc.) or, in marshalled form, inside a state store.

All interactions between crash-only components should have a timeout. This includes explicit communication as well as RPC: if no response is received to a call within the allotted timeframe, the caller assumes the callee has failed and reports it to a recovery manager, which crash-restarts the callee (thus enforcing the assumption of failure). Crash-restarting helps ensure the called component is in a known state; this is safe because the component is crash-safe and crash-restart is idempotent. Timeouts combined with crashing provide an orthogonal mechanism for turning all non-Byzantine failures, both at the component level and at the network level, into fail-stop events (i.e., the failed

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

entity either provides results or is stopped), even though the components are not necessarily fail-stop. Such behavior is easier to accommodate, and containment of faults is improved.

4.3.4 Component/request decoupling

While inter-component decoupling enables *structural* reintegration of a microrebooted component after recovery, we also need *functional* reintegration. This is achieved via retriable requests, that decouple components from the requests they process.

When a component invokes a currently microrebooting component, it receives from the application platform (OS, application server, etc.) a `RetryAfter(t)` exception; the call can then be re-issued after the estimated recovery time t, if it is idempotent. For non-idempotent calls, rollback or compensating operations can be used, in order to make these calls idempotent. Making all requests idempotent can significantly simplify recovery. If components transparently recover in-flight requests this way, intra-system component failures and microreboots can be hidden from end users.

Component/request decoupling is achieved by virtue of the fact that any request that failed due to component-level recovery can be retried on a different, non-failed instance of that component. Requests, however, must make the state and context needed for their processing explicit. This allows a fresh instance of a rebooted component to pick up a request and continue from where the previous instance left off. Requests also carry information on whether they are idempotent (to indicate whether the request can be retried transparently or not) along with a time-to-live (to avoid endless retries). Both idempotency and TTL information can initially be set at the system boundary, or by the client. For example, the TTL may be determined by load or service level agreements, and idempotency flags can be based on application-specific information (which can be derived, for instance, from URL substrings that determine the type of request). Many interesting operations in an Internet service are idempotent, or can easily be made idempotent by keeping track of sequence numbers/timestamps or by wrapping requests in transactions; some large Internet services have already found it practical to do so [Pal02]. Over the course of its lifetime, a request will split into multiple sub-operations, which may rejoin, in much the same way nested transactions do.

4.3.5 Component/resource decoupling

One can view explicit deallocation of resources as a mere optimization, not a required condition for the program to run properly. Should resources not be explicitly deallocated, they must be released by the application platform whenever needed in order to continue correct execution.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Having decoupled components from each other and from the requests they process, the last step is to decouple components from the resources needed to process the requests. Resources in a frequently-microrebooting system should be leased [GC89], to improve the reliability of cleaning up after a microreboot, which may otherwise leak resources.

In addition to memory and file descriptors, we believe certain types of persistent state should carry long-term leases; after expiration, this state can be deleted or archived out of the system. One example is account profiles for a free e-mail provider: every time the user logs in, a 6-month lease is renewed; when the lease expires, all associated data can be purged from the system. One can also imagine leasing CPU resources: if a computation is unable to renew its execution lease, it is terminated by a high confidence watchdog [Fet03]. If requests carry a time-to-live, then stuck requests can be automatically purged from the system once this TTL runs out. For example, in PHP, a server-side scripting language used for writing dynamic Web pages, runaway scripts are killed and an error is returned to the Web browser. Leases give us the ability to reason about the conditions that hold true of the system's resources after a lease expires. Infinite timeouts/leases are not acceptable; the maximum-allowed timeout and lease are specified in an application-global policy. This way it is less likely that the system will hang or become blocked.

4.4 Discussion

The crash-only design approach embodies well-known principles for robust programming of distributed systems. We push these principles to finer levels of granularity within applications, giving non-distributed applications the robustness of their distributed brethren, along with the ensuing benefits and drawbacks.

Building crash-only systems is not easy; the key to widespread adoption of our approach will require employing the right architectural models and having the right tools. With the recent success of component-based architectures (e.g., J2EE and .NET), and the emergence of the application server as an operating system for Internet applications, it is possible to provide many of the crash-only properties in the platform itself, as we show in chapter 6. This would allow all applications running on that platform to take advantage of the effort and become crash-only.

When counting on request retry to hide unavailability from end users, the finer the grain of these requests, the more successful the result, because the effect on service quality is smaller (see section 8.6). We therefore focus on applications whose workloads can be characterized as relatively short-running tasks that frame state updates. Substantially all Internet services fit this description,

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

in part because the nature of HTTP has forced designers into this mold. As enterprise services and applications (e.g., workflow, customer management) become Web-enabled, they adopt similar architectures, thus widening the spectrum of applications that are amenable to microreboot-based recovery.

In order for a crash-only system to make reasonable progress, enough of its requests must be idempotent to avoid frequent rollbacks. This requirement might be inappropriate for some applications. Our proposal does not handle Byzantine failures or data errors, but such behavior can be turned into fail-stop behavior using well-known orthogonal mechanisms, such as triple modular redundancy [GR93] or clever state replication [CL99].

In today's Internet systems, fast recovery is obtained by overprovisioning and counting on rapid failure detection to trigger failover. Such failover can sometimes successfully mask hours-long recovery times, but often detecting failures end-to-end takes longer than expected. Crash-only software is complementary to this approach and can help alleviate some of the complex and expensive management requirements for highly redundant hardware, because faster recovering software means less redundancy is required. In addition, a crash-only system can reintegrate recovered components faster, as well as better accommodate removed, added, or upgraded components.

We expect throughput to suffer in crash-only systems, but this concern is secondary to the high availability and predictability we expect in exchange. The first program written in a high-level language was certainly slower than its hand-coded assembly counterpart, yet it set the stage for software of a scale, functionality and robustness that had previously been unthinkable. These benefits drove compiler writers to significantly optimize the performance of programs written in high-level languages, making it hard to imagine today how we could program otherwise. We expect the benefits of crash-only software to similarly drive efforts that will erase, over time, the potential performance loss of such designs.

The dynamics of loosely coupled systems can sometimes be surprising (e.g., unforeseen synchronization between Internet router updates [FJ94]). Resubmitting requests to a component that is recovering can overload it and make it fail again; for this reason, the *RetryAfter* exceptions should provide an estimated time-to-recover. This estimated value can be used to spread out request resubmissions, by varying the reported time-to-recover estimate across different requesters. A maximum limit on the number of retries may be specified in an application-global policy, along with the lease durations and communication timeouts. These numbers can be dynamically estimated based on historical information collected by a recovery manager [CKK+03], or simply captured in a static description of each component, similar to deployment descriptors for EJBs. In the absence of such

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

hints, a simple load balancing algorithm or exponential backoff can be used.

There is also a natural tension between the cost of restructuring a system to make it crash-only and the cost (in downtime) of not restructuring it. Fine module granularity improves the system's ability to tolerate partial restarts, but requires the implementation of a larger number of internal, asynchronous interfaces. The paradigm shift required of system developers could make such a design too expensive in practice and, when affordable, may lead to buggier software. In some cases crash-only design may simply not be feasible, such as for systems with inherent tight coupling (e.g., real-time closed-loop feedback control systems).

4.5 Chapter Summary

This chapter proposed a design that enables the systematic use of (micro)reboots to recover from failures. Crash-only programs crash safely and recover quickly; there is only one way to stop such software – by crashing it – and only one way to bring it up – by initiating recovery. Crash-only systems are built from crash-only components, and the use of transparent component-level retries hides intra-system component crashes from end users.

We took the view that transient failures will continue plaguing the software infrastructures we depend on, and thus reboots are here to stay. We proposed turning the reboot from a demonic concept into a reliable partner in the fight against system downtime, given that it is a time-tested, effective technique for circumventing Heisenbugs.

By using a crash-only approach to building software, we expect to obtain better reliability and higher availability. Application fault models can be simplified through the application of externally-enforced "crash-only laws," thus encouraging simpler recovery routines which have higher chances of being correct. Writing crash-only components may be harder, but their simple failure behavior can make the assembly of such components into large systems easier. The promise of a simple fault model makes stating invariants on failure behavior possible. A system whose component-level and system-level invariants can be enforced through crash-rebooting is more predictable, making recovery management more robust. In chapter 7 we quantitatively evaluate these hypotheses.

Building crash-only systems in a systematic way requires a framework consisting of well-understood design rules. Our attempt at formulating such a framework was presented here, advocating the paradigm of building applications as distributed systems, even if they are not distributed in nature.

While reboot-based recovery is still a coarse recovery method in today's systems, in the next

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

chapter we will describe how this recovery sledgehammer can be turned into a scalpel, as long as the system being recovered is crash-only.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 5

Microreboot-based Recovery

In this chapter we show that, with a small number of changes, runtime platforms can support a mechanism by which fine-grained components of crash-only applications are recovered through microreboot at the first indication of failure. In later chapters, we will demonstrate that microrebooting can achieve many of the same benefits as a process restart in Java systems, while reducing unavailability by a factor of 50. The combination of crash-only software and microrebooting is a better cost/dependability tradeoff compared to the approach of aiming for correct code and supporting diverse recovery mechanisms. Here we describe in general terms the infrastructure needed to recover a crash-only application using microreboot, along with a simple policy that tolerates failure of the recovery mechanism itself.

5.1 The Microreboot Mechanism

The model used throughout this dissertation is that of a crash-only application (consisting of crash-only components) running atop an execution platform that has complete control over the execution of the application. Such a platform could be an operating system running user programs, an application server running hosted applications, an object framework running a distributed program, etc. In this dissertation, the system in question can be any Java 2 Enterprise Edition (J2EE) application running on a suitably modified J2EE application server (see chapter 6); individual EJBs (application components) can be microrebooted in response to actual failure or just hints of impending failure.

Focusing on application availability is motivated by the fact that most downtime-causing failures in large-scale business-critical systems occur at the application level [Mit04, Cho03, Pal02, Rei04]. Applications are therefore the weakest link, while hardware, operating systems, databases, Web

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

servers are much more reliable, by comparison. This is not surprising, given that most business value is created by applications, not the execution infrastructure itself; as a result, it is the applications that change most often and evolve the fastest. We can therefore reliably place support for microrebooting in the execution platform itself.

The model is abstractly represented in figure 5.1. The execution platform, be it an operating system, application server, or Java virtual machine, instantiates and runs applications built from components (shown as `COM` units), which interact with each other and with the platform's services. These components can be any of a variety of types: threads, processes, Java beans, .NET services, etc. We extend the execution platform with a microreboot facility that performs the recovery per se, and a failure monitoring facility (both are described in section 5.1.1).

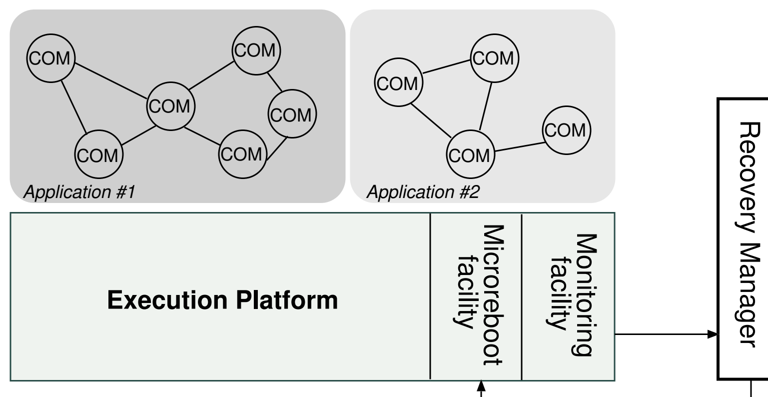

Figure 5.1: A crash-only application running atop a microreboot-enabled execution platform.

The monitoring agents are in charge of supervising the health of the applications and reporting interesting changes to an external recovery manager. The manager decides which components need to be recovered, thus implementing the desired recovery policy. These decisions are transmitted to the microreboot facility for execution. Should the recovery not cure the observed failure, we progressively enlarge the recovery perimeter to the larger, containing subsystem, as described in section 5.2.2. The recovery attempts continue with increasingly larger scope, attempting to find the fault recursively through successively broader fault domains until the failure stops manifesting or until human intervention is deemed necessary; more details can be found in section 5.2.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

5.1.1 Platform-level support

The execution platform is a piece of software in control of the application components' lifecycle: instantiation/destruction, starting/stopping, and suspending/resuming (for example, Java application servers can perform these operations on Java components). To this platform we add a `microreboot()` facility, described by the pseudocode shown in figure 5.2.

```
microreboot( component c )
    reject new requests to c with MicrorebootInProgressException(t)
    terminate requests that are in progress at c
    release c's resources
    destroy c
    initialize new instance of c
    integrate new c' into running system
    resume accepting requests to c'
```

Figure 5.2: Pseudocode for a generic microreboot facility.

The role of the microreboot facility is to cleanly remove the faulty component from the system, restart it, then reintegrate it into the running system. For this to be effective, the execution platform must

- mediate all interactions between the application's components, in order to control the suspension and resumption of new requests (e.g., in a Java application server, application components call each other over RMI, Java's RPC-like remote method invocation service provided by the application server)
- control the resources of application components, so they can be promptly reclaimed upon microreboot (e.g., an operating system kernel has this type of control over the local resources of processes)

5.1.2 State stores

As described in chapter 4, specialized state stores are an important ingredient of any crash-only system. We gave as examples databases, session state managers, and persistent hashtables. Since the state stores are themselves crash-only, they are amenable to reboot-based recovery as well. We therefore hook state store recovery into the recovery manager, and treat the state stores as just another set of components (albeit which fail much less frequently).

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

5.1.3 Correctness of microrebooting

We will argue in this section that a system that recovers by microrebooting provides no less correctness than if it recovered by full rebooting. For this, we need to show that a wrong response provided by a correct crash-only system cannot be attributed to having recovered from a failure by microreboot instead of a full reboot in the system's past. Correct behavior in the face of microreboots is difficult to define in the absolute, since (by assumption) microreboots are performed in response to a failure; with or without a microreboot, such failure might cause incorrect behavior.

Impact on shared state

We have defined critical state as state that can only be recreated by replaying the end user interaction with the system; all such state is stored in dedicated state stores in a crash-only system. The result of a particular user request depends only on the components it calls and on any critical state that is involved in answering the request (whether or not that state is directly visible outside the component boundary). Instances of components that are stateless (with respect to critical state) are by definition indistinguishable from each other.

Since all critical shared state resides in external state stores, a microreboot is indistinguishable from a full reboot: the state store itself cannot distinguish a microreboot from a full one, so consistency of the stored state is not changed. For example, if the state resides in a session state store like SSM, then the object representing the session state is always stored atomically and in its entirety – microrebooting a component engaged in a SSM update looks to SSM just like a full reboot.

If critical state resides in several state stores and updates to this state are made in a non-atomic fashion, then microrebooting one component may leave that state inconsistent, without notifying the other component(s) that share it. A full reboot, on the other hand, would restart all components simultaneously, thus not giving them an opportunity to read the inconsistent state.

First, we advocate that the implementation of the "strong boundaries" requirement of crash-only design prevent such scenarios (e.g., in Java components, object references should not be passed between components and the use of static variables should be conservative; these "best practices" could be enforced by a suitably modified JIT compiler). When the runtime detects the presence of such unsafe state sharing practices, it should disable the use of microreboots for the application in question.

Second, in addition to prevention through language mechanisms, it is also possible to declare higher level application invariants on shared critical state as constraints and assertions in the data

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

schema at the various state stores. Mechanisms such as J2EE's container-managed persistence offer the option of describing these invariants in the deployment descriptors and having the runtime enforce those constraints transparently, rather than requiring the state stores to do it.

Third, mechanisms such as that introduced in section 6.3.5 and described in appendix A, can help determine groups of components that must be microrebooted together in order to preserve correctness.

In addition to refreshing all components, a full reboot also discards the volatile shared state, regardless of whether it is inconsistent or not; microrebooting allows such state to persist. In a crash-only system, state that survives the recovery of components resides in a state store that assumes responsibility for data consistency. In order to accomplish this, dedicated state repositories need APIs that are sufficiently high-level to allow the repository to repair the objects it manages, or at the very least to detect corruption. Otherwise, faults and inconsistencies perpetuate; this is why application-generic checkpoint-based recovery in UNIX was found not to work well [LCC00]. In the logical limit, all applications become stateless and recovery involves either microrebooting the processing components, or repairing the data in state stores.

An interesting case is that of necessary, but non-critical state, that is lost upon microrebooting and could be recreated post-microreboot, but isn't. For example, a component may build up information about how many other components are running in order to estimate system capacity and do admission control; upon microreboot, this information could be lost and not recreated. If the system were rebooted entirely, then the system would start in the original configuration and the capacity estimator would not need updated configuration information. Such a design is in violation of the crash-only principles, because a restart-after-crash behaves differently from a regular start; capacity information should be reconstructed at each start (i.e., start=recovery).

Interactions with external resources

If a component is able to circumvent the execution platform in the acquisition of an external resource that the platform is not aware of, then microrebooting the component may leak the resource in a way that a full reboot would not. For example, a Java component X, running on a J2EE application server, could directly open a connection to a remote database without using the application server's transaction service, acquire a database lock, then share that connection with another component Y. If X is microrebooted prior to releasing the lock, Y's reference will keep the database connection open even after X's recovery, and thus X's DB session stays alive. The database will not release the lock until after X's DB session times out. In the case of a full reboot, however, the resulting

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

termination of the underlying TCP connection by the operating system would cause the immediate termination of the DB session and the release of the lock. If the application server knew that X acquired a DB session, it could properly free that session when X is microrebooted.

It is for this type of reasons that we require application components to obtain resources exclusively through the facilities provided by their platform.

Microrebooting is well-suited for Internet services

We believe the microreboot technique is best suited for large-scale Internet services and any applications that fit this mold, such as recent enterprise applications. The workloads faced by such services consist of short-lived, mostly-independent requests coming from a large population of distinct users. The work that Internet services must do is generally partitioned into disjoint sets of discrete operations; microreboots take advantage of the application's structure to realize this potential.

Additionally, the underlying protocol (HTTP) and most of the application logic is stateless and, except for marked, non-idempotent requests, end-users can safely retry failed requests until they succeed. This lets us microreboot components in the system, knowing that any users affected will face only a minor inconvenience. In fact, this property makes it useful to recover even from purely deterministic bugs such as a pathologically malformed request: if recovery is fast enough, other users issuing non-pathological requests may still be able to use the service.

Many Internet services today use large in-memory caches in order to avoid the bottleneck of central databases (e.g., the servers at a large Internet portal use 64 GB of RAM just for caching database queries [Pal02]). Unfortunately, a full machine reboot flushes this cache, and re-warming it can take a long time: transferring 64 GB over a 40 MB/sec SCSI bus takes on the order of half an hour, which is why whole-system reboots are generally avoided.

Microrebooting can be viewed as an optimization over rebooting, because any failure that can be cured with a microreboot could have been cured with a full reboot – the difference is that microrebooting can do so more than an order of magnitude cheaper (chapter 7). However, as will be shown in chapter 8, microrebooting goes beyond just optimization, as it enables a qualitative change in the way failures are handled.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

5.2 Microreboot-based Recovery Policy

Microreboots separate the concern of recovery from that of diagnosis and bug finding. When an online system fails, downtime is expensive and the first priority is to restore service by any means available. Identifying and fixing the root cause of the transient failure is a separate effort, and microrebooting does not aid this effort in a direct way, nor does it provide much more than a "temporary fix." Thorough logging mechanisms help developers fix root causes.

5.2.1 Recovery groups

When the recovery manager receives a failure notification from the monitoring agents, it must decide which component to microreboot. In the ideal case, each component is individually recoverable; in practice, this is not always possible due to dependencies between components. Such dependencies can be:

- inherent to the application, such as a *Login* component requiring an *Authenticate* component to be present in order to complete a user login;
- due to lack of programmer discipline (such as the inappropriate use of shared state) or due to legacy software that is too costly to partition into components;
- due to idiosyncratic interactions (for example, *UpdateUser* may leak database connections, thus causing another component to fail due to the unavailability of such connections).

A representation of these dependencies is necessary for effective microreboot-based recovery; we call such a representation a failure propagation map (f-map). An f-map is a directed graph whose nodes are the application components and whose edges represent propagation paths of faults from one component to another (see figure 6.5 for an example). Note that f-maps are not guaranteed to be correct, but rather serve as an aid to improve overall recovery effectiveness.

The transitive closure of a component c over the f-map is called a recovery group, and represents the unit of microreboot recovery. In the presence of dependencies, when microrebooting a component c is necessary, the corresponding recovery group must be microrebooted. This ensures both correctness (by avoiding inconsistencies) as well as recovery performance (by avoiding causing other failures that will need to be detected, localized, and recovered later).

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

5.2.2 Recursive microrebooting

Microrebooting a set of components may not cure the observed failure, either because

- some error has propagated from faulty components to their neighbors outside the recovery group, along an edge not captured in the current instance of the f-map, or
- the failure is not reboot-curable (e.g., a persistent failure, a condition triggered deterministically by bad input), or
- the recovery manager chose the wrong component to microreboot.

For these situations, we introduced the notion of *recursive microrebooting*: upon noticing that a microreboot has not cured the failure as expected, the recovery manager initiates progressively coarser grained restarts until either the failure no longer manifests, or the top of the system hierarchy has been reached. In the latter case, the recovery manager either notifies a system administrator (by pager, email, etc.) or invokes an alternate recovery mechanism. As will be seen in chapter 7, microreboots are generally cheap enough to attempt them as a first-line recovery prior to any other recovery mechanism.

An example of such recursive microrebooting would be in a Java system, where individual components are microrebooted, then larger groups of components, then the entire JVM, and finally the operating system. If a full machine reboot does not eliminate the failure symptoms (due, for instance, to a failed disk drive), then a human operator is notified.

The recovery manager also recognizes repeating patterns in order to prevent infinite microreboot loops. If two or more components form an undetected reboot-failure cycle, then microrebooting one component could cause the second to fail, and rebooting the second component causes the original component to fail again, repeating the process endlessly. To identify such patterns, the recovery manager records a signature tuple $\sigma =$ <*symptom, suspected-faulty components, action taken*> in a history log and, upon receiving a new failure symptom, looks it up. If it reacted to the same symptom within a recent Δt amount of time, it immediately enlarges the scope of recovery, otherwise it performs the requisite microreboot.

In the same vein, the recovery manager tracks its past responses to prior failure symptoms to optimize recovery. If a symptom is received that required multi-level recursive microrebooting in the past, the recovery managers short-circuits this process and performs only the final level of recovery.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

5.2.3 Tradeoffs

A detailed analysis of the various recovery policies and tradeoffs involved is beyond the scope of this dissertation; nevertheless, in this section we touch upon some of the issues that microreboots bring to light. We have purposely separated mechanism from policy in the above design in order to allow new policies to be seamlessly plugged in, to encourage further research on the topic.

Knowing *what* to microreboot

Microrebooting is an application-generic recovery technique for componentized applications, which can be supported entirely in the execution platform, and requires no recovery awareness in the application other than the crash-only software principles. It promises a reduction in recovery time, but it also has a reduced certainty of success. In the prototype described in chapter 6, microrebooting is 1-2 orders of magnitude faster than a full restart – in this case, the reduced certainty of success is of little consequence, since the cost of attempting a microreboot is minimal.

This, however, may not be the case in all systems. Should the relative benefit of a microreboot be considerably lower, identifying the exact components that need to be recovered becomes increasingly more important to reducing recovery time. Since we advocate avoiding the pollution of the recovery manager with a priori knowledge of application structure, it can be claimed that recursive microrebooting is an application-generic recovery approach. Yet, in the face of incomplete application information, optimizations are difficult.

Resolving this dilemma is system-specific. Chapter 6 describes our approach in a Java environment; others can build upon these initial constructions and extend to other environments as well.

Preventing failures and curbing fault propagation

By microrebooting promptly at the first indication of a failure, faults are prevented from propagating to the rest of the system. Such faults typically propagate through calls between healthy and faulty components; prompt recovery prevents some of these contaminating calls. Furthermore, we can microreboot prior to any observed failure, as a way to prevent such failure from occurring.

Based on the f-map and monitoring information, the recovery manager has the ability to not only make reactive recovery decisions but proactive preventive maintenance decisions as well. Software rejuvenation [HKKF95, GHKT96] has been shown to be a useful technique for staving off failure in systems that are prone to aging; for instance, rebooting several times a day Apache Web servers that leak memory is an effective way to prevent them from failing [Bre01b]. The recovery manager

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

in a recursively recoverable system tracks components' failure histories and infers for how long a component can be expected to run without failing due to age; restarting it before that time runs out will avert aging-related failure. The observation of fail-stutter behavior [ADAD01b] can also trigger rejuvenation. A number of sophisticated models have been developed for the software aging process [GPTT95, GMVT98], but experience with deployed large scale Internet services seems to indicate that simple observation-based strategies work best [Bre01b].

Similarly, the recovery manager can identify statistical correlations between the failure of one component (or some other types of events) and the subsequent failure of other component(s); should the former be observed, recovery of the latter can be triggered preventively.

Delaying a full reboot

The more state gets segregated out of the application, the less effective a reboot becomes at scrubbing this data. When a full process restart is required, poor diagnosis may result in one or more ineffectual component-level microreboots. As discussed in Section 8.1, failure localization needs to be more precise for microreboots than for full restarts. Using the recursive policy, microrebooting progressively larger groups of components will eventually restart the entire system, but later than could have been done with better diagnosis.

If we think of the recursive microrebooting policy as being described by an abstract "microreboot tree," it becomes clear that, the closer to the bottom a restart occurs, the less expensive the ensuing downtime, but the lower the confidence that transient failures will be resolved. In the extreme, a full reboot is the last action taken (root of the tree), but it is certain to fix all reboot-curable failures. When a failure manifests, a sophisticated recovery manager could use a cost-of-downtime/benefit-of-certainty analysis to decide at what level of granularity to microreboot.

This same set of tradeoffs can be utilized in tuning the proposed rejuvenation regimen: it could be as simple as rebooting periodically, or as sophisticated as a differentiated restart treatment for each subsystem/component based on various parameters and variables. Identical systems can have different revival and rejuvenation policies, depending on the application's requirements and the environment they are in. Scheduled non-uniform rejuvenation can transform unplanned downtime into planned, shorter downtime, and it gives the ability to more often rejuvenate those components that are critical or more prone to failure. For example, a recent history of revival restarts and load characteristics can be used to automatically decide how often each component requires rejuvenation. Simpler, coarse-grained solutions have already been proposed by Huang et al. [HKKF95] and are used by IBM's xSeries servers [Int01].

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

5.3 Chapter Summary

Adopting a model in which recovery is performed by microrebooting can be conducive to more robust software even in the absence of performing said recovery.

The unannounced restart of a software component is seen by all other components as a temporary failure; systems that are designed to tolerate such restarts are inherently tolerant to all transient non-Byzantine failures. Since most manifest software bugs and hardware problems are transient [MMS$^+$00, MD99, Cho97], a strategy of failure-triggered, reactive component restarts will mask most faults from the outside world, thus making the system as a whole more available.

Since our approach is based on observation and control at the platform layer, it is application-generic and requires no a priori knowledge of application structure. This addresses the fact that today's services are heterogeneous and dynamic, encompassing many vendors' hardware and software components that evolve rapidly and often turbulently. We can therefore cast observed failures into microreboot-curable failures, and only treat them as special failures when microrebooting does not cure them.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 6

Prototype and Experimental Setup

In evaluating microreboot-based recovery, we aimed to construct a testbed that is as close to a real Internet service as a lab environment can permit. A common design pattern for Internet-connected applications is the three-tiered architecture [Jac03, CMZ02]: a presentation tier consists of stateless Web servers, the application tier, and the back-end tier. The last decade has seen a number of systems, such as those hosting enterprise applications, migrate to the three-tiered architecture. The presentation tier handles and demultiplex incoming HTTP connections, the application logic tier runs the code that constitutes the application, and the back-end tier stores persistent data.

We chose to use the enterprise edition of Java (J2EE) [Sun], a component framework specifically designed to simplify the development of large-scale enterprise applications in the three-tiered model. Motivated by J2EE's popularity (40% of the current enterprise application market [Bar04]), we chose to add microreboot capabilities to the most widely used open-source J2EE application server (JBoss) and converted a J2EE application (RUBiS) to the crash-only model. The changes we made to the JBoss platform universally benefit all J2EE applications running on it.

We start this chapter by describing the details of J2EE (section 6.1) and the synergies between this programming platform and the crash-only principles (section 6.2). We then describe the testbed itself, including all the extensions we made to JBoss and other software used in our experiments (section 6.3). We present our failure detection and localization mechanism (section 6.4), the client emulator (section 6.5), and finally the action-weighted throughput metric used for the evaluation (section 6.6).

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

6.1 Overview of J2EE

J2EE is an extension of the Java language and runtime, allowing Java programmers to seamlessly use facilities that, previously, required complex programming; examples include access to remote databases and transactional control, LDAP and other authentication and naming infrastructures, interactions with SOAP and WSDL-based Web services, CORBA object brokers, etc. In spite of a number of shortcomings, J2EE has largely succeeded in allowing a wider set of programmers to create Web-accessible enterprise applications that are portable between platforms and scalable, while integrating with legacy technologies.

J2EE applications consist of portable Java components, called Enterprise Java Beans (EJBs), and platform-specific XML deployment descriptor files. J2EE applications are hosted by and run on an application server, as shown in figure 6.1. The J2EE application server, akin to an operating system for Web-connected enterprise applications, uses the deployment information to instantiate an application's EJBs inside management containers; there is one container per EJB object, and it manages all instances of that object. The server-managed containers provide the application components with a rich set of services: thread pooling and lifecycle management, client session management, database connection pooling, transaction management, security and access control, etc. A J2EE application is able to run on any J2EE application server, with modifications only needed in the deployment descriptors.

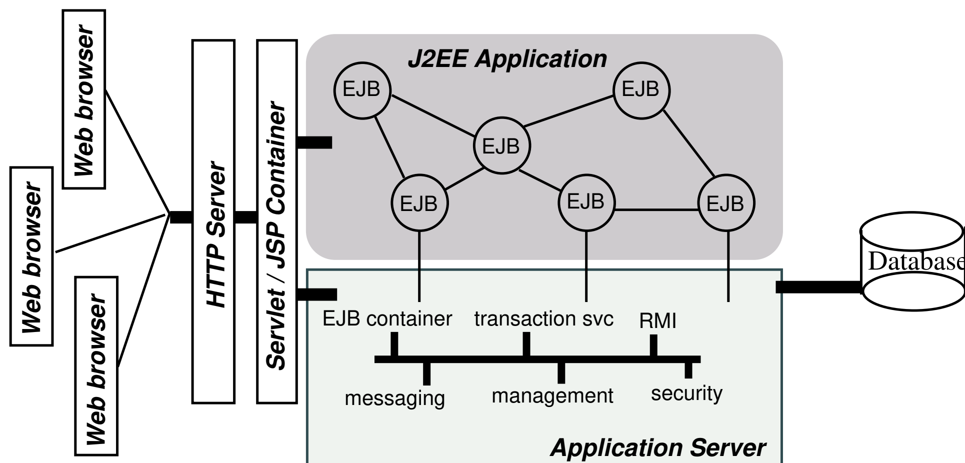

Figure 6.1: Architectural diagram of a J2EE application environment.

End users interact with a J2EE application through a Web interface, the application's presentation tier, encapsulated in a WAR – Web ARchive. The WAR component consists of servlets and

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Java Server Pages (JSPs) hosted in a Web server that resides outside the application server. The Web tier translates URL accesses into invocations of EJB methods and then formats the returned results into HTML pages for presentation to the end user.

To run a J2EE application, one must boot the operating system, start the J2EE application server, start any necessary additional components required by the application (e.g., a database used for persistent state storage and the Web server front-ends), and finally "deploy" the application on the application server, i.e. instantiate each EJB in its container and allow the application to begin accepting requests from the Web servers.

When the application server receives a call from the Web tier, it retrieves an idle thread from its thread pool, associates it with the request, and then allows the thread to carry the request through the various EJBs that need to be invoked. Invoked EJBs can call on other EJBs, interact with the back-end databases, invoke other Web services, etc. An EJB is similar to an event handler, in that it does not constitute a separate locus of control – a single Java thread shepherds a user request through multiple EJBs, from the point it enters the application tier until it returns to the Web tier.

EJBs provide a level of componentization that is suitable for building crash-only applications; this, together with the wide accessibility of the J2EE platform, were the primary reasons we chose J2EE for our prototype. Aside from JBoss, there are a number of other application servers to which all the modifications described here could be applied: Weblogic from BEA, Websphere from IBM, iAS from Oracle, JRun from Macromedia, JOnAS from the ObjectWeb Consortium, Geronimo from the Apache Software Foundation, Java AS from Sun, Enterprise Server from Borland, Resin from Caucho, etc. The principles of microreboot-based recovery also easily carry over to frameworks like Microsoft's .NET, as well as the LAMP architecture (Linux, Apache Web server, MySQL database, PHP/Python/Perl) for developing Web applications.

6.2 Synergies with Crash-Only Design

As described in section 4.3, to write a crash-only application in J2EE, we would need the J2EE application server to provide strong isolation between components, and these components must be fine-grained. There must be a way to segregate critical state from the application into dedicated state stores. We must also be able to decouple components from each other, decouple components from the requests they handle, and from the resources they use.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

6.2.1 Component management

EJBs, the J2EE components, run in the managed environment of an application server, which provides containers in which the beans are instantiated and run. The application server, aided by the type safety of the Java language, provides clear boundaries around these components. Most J2EE application servers provide the ability to deploy/undeploy individual EJBs, which offers sufficient structure in the server for implementing microreboots.

The Web server processes that dispatch incoming HTTP traffic to EJBs are also self-contained and can be managed independently of the EJBs.

6.2.2 Decoupling

Whenever an EJB wants to invoke another EJB's method, it looks up the target EJB by name in the Java Naming Directory (JNDI, provided by the application server) and uses the Java class resulting from the lookup to make the invocation (similar to the way RPC stubs work). The inter-EJB calls themselves are also mediated by the application server via the containers, to abstract away the details of remote invocation (if the application server is running on a cluster) or replication (if the application server has replicated a particular EJB for performance or load balancing reasons).

Since doing a lookup on every call is expensive, JBoss provides the caller with a proxy on the first lookup, which then handles all subsequent calls without interacting with JNDI. This model ensures that inter-bean communication is mediated by the application server, which is particularly useful when an EJB is in the process of being microrebooted – with suitable modifications in the application server, we can cleanly expose the recovery either by throwing back an exception to the caller or by stalling the call until the target component has recovered.

We recommend against the use of direct references that span the boundaries of components. Indirect, microreboot-safe references can be maintained outside the components, either by a state store or by the application platform. For EJBs that do maintain references to other EJBs, microrebooting a particular EJB causes those references to become stale. To remedy this, whenever an EJB is microrebooted, we also microreboot the transitive closure of its inter-EJB references (the recovery group), as described in section 5.2.1. This ensures that, when a reference goes out of scope, the referent disappears as well.

Further containment of recovery is obtained through compiler-enforced interfaces and type safety. EJBs cannot name each others' internal variables, nor can they use mutable static variables. While this is not enforced by the compiler, J2EE "best practices" documents warn against

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

the use of static variables and recommend instead the use of singleton EJB classes, whose state is accessed through standard accessor/mutator methods.

6.2.3 State segregation

Web-enabled applications, like the ones we would expect to run on JBoss, typically handle three types of important state: long-term data that must persist for years (such as customer information), session data that needs to persist for the duration of a user session (e.g., shopping carts or workflow state in enterprise applications), and virtually read-only data (static images, HTML, JSPs, etc.).

There are three types of EJB: (a) entity beans, which map each bean instance's state to a row in a database table, (b) session EJBs, which are used to perform temporary operations (stateless session beans) or represent session objects (stateful session beans), and (c) message-driven EJBs (not of interest to this work). EJBs may interact with a database directly and issue SQL commands, or indirectly via an entity EJB. In microrebootable applications we require that only stateless session beans and entity beans be used; this is consistent with best practices for building scalable EJB applications. The entity beans must make use of Container-Managed Persistence (CMP), a J2EE mechanism that delegates management of entity data to the EJB's container. CMP provides relatively transparent data persistence, relieving the programmer from the burden of managing this data directly or writing SQL code to interact with a database. Our prototype applications conform to these requirements.

Session state must persist at the server for long enough to synthesize a user session from independent stateless HTTP requests, but can be discarded when the user logs out or the session times out. Typically, this state is maintained inside the application server and is named by a cookie accompanying incoming HTTP requests. To ensure the session state survives both microreboots and full reboots, we externalize session state into a session state store that we modified to integrate well into a J2EE environment. Many commercial application servers forgo this separation and store session state in local memory only, in which case a server crash or EJB microreboot would cause the corresponding user sessions to be lost.

The segregation of state offers a certain degree of recovery containment, since data shared across components by means of a state store does not require that the components be recovered together. Externalized state also helps to quickly reintegrate recovered components, because they do not need to perform data recovery following a microreboot.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

In summary, the J2EE application model is not completely crash-only, but offers a good compromise for the requirements of microrebooting. We now describe the particular J2EE application server implementation that we augmented for the work in this paper.

6.3 Testbed

To evaluate microreboot-based recovery, we modified JBoss, developed session state stores, developed a client emulator, a fault injector, and a system for automated failure detection, diagnosis, and recovery. Figure 6.2 schematically describes the connections between these various components – all of these will be described in subsequent sections.

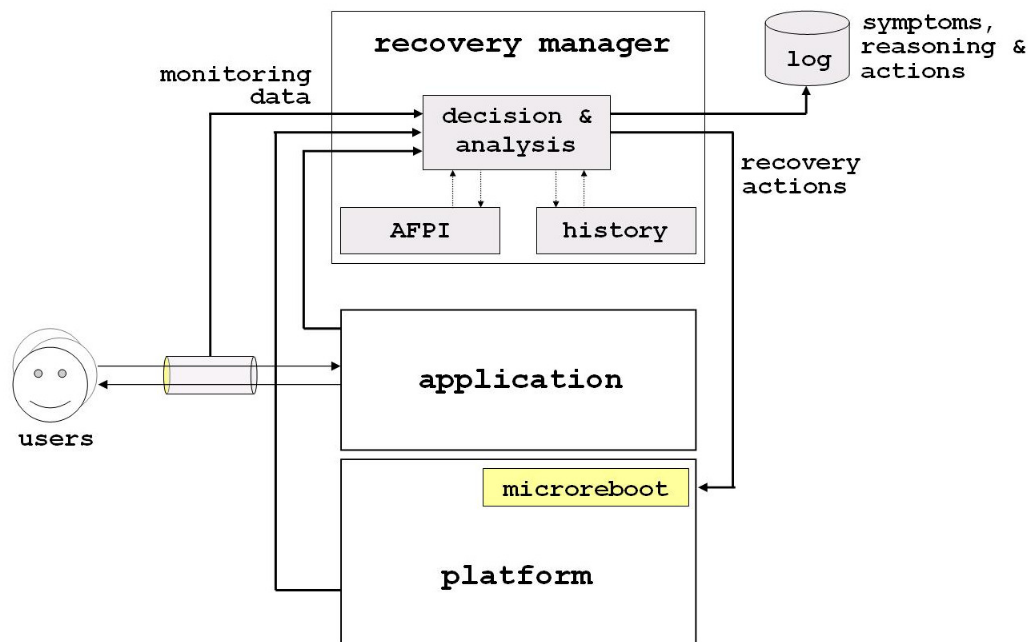

Figure 6.2: Conceptual architecture of our J2EE prototype.

6.3.1 A microreboot-enabled application server

We built a platform for crash-only applications by extending JBoss [JBo02], an open-source application server that complies to the J2EE standard. JBoss's performance and features compare favorably with proprietary closed-source offerings [Bar04]. It received early on the JavaWorld 2002 Editors' Choice Award over several commercial competitors, and has since been downloaded from

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

SourceForge several million times. More than 100 corporations, including WorldCom and Dow Jones, are using JBoss for demanding computing tasks.

We added a microreboot method to JBoss, that can be invoked programatically from within the server, or remotely by the recovery manager, over HTTP. Since we modified the JBoss server, microreboots can now be performed on any J2EE application; however, this is safe only if the application is crash-only. The microreboot method can be applied to one or more EJB or WAR components.

Pseudocode of our JBoss implementation of microreboot appears in figure 6.3.

```
microreboot_JBoss( component c )
    reject new requests to c with MicrorebootInProgressException(t)
    kill threads associated with instances of c
    destroy all instances of the c object
    release all c resources known to JBoss
    destroy c's container
    release all metadata maintained by JBoss on behalf of c
    replenish pool of worker threads
    re-instantiate a container for c
    create new instance of c in new container
    invoke c's start() method
```

Figure 6.3: Pseudocode for the JBoss implementation of microreboot.

1. The server starts off by suspending the relaying of any new requests to the microrebooting component. Since all inter-EJB calls pass through the component containers, and the application server has control over these containers, this change fits cleanly in the existing architecture. Any call to this component (while microrebooting) will result in a special *MicrorebootInProgressException(t)*, which can be handled by the caller's container transparently to the caller (e.g., by retrying after t msec), or can be propagated to the caller code itself. t is the estimate, in milliseconds, of how much longer it will take for the callee to recover; the callee's container could add a randomized Δt to the value of t, in order to avoid overload upon completion of recovery.

   Inside the callee's container, we set a flag indicating the callee is currently microrebooting. Any time a call comes in, we check this flag and only pass the call to the component code if the flag is not set.

   We have disabled transparent call retries in all experiments reported in chapter 7, in order to

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

avoid masking the downtime induced by a microreboot.

2. All worker threads that are associated with instances of the microrebooted component are killed. This will automatically interrupt all request processing within the microrebooted component and the callers either time out or receive exceptions, both being cases that can be handled cleanly by the caller's container or the caller code itself. Calls that were in-progress at the time of the microreboot will fail in exactly the same way they would fail if the component crashed, had a bug, etc.

3. A component is a Java object, and as such has several instances; in a production system there could be hundreds to many thousands of instances of the same component. As part of the microreboot, we destroy all these instances.

   It is worth noting that this is a coarse-grained version of a microreboot – if we only destroyed the faulty instance(s), then the results reported in chapter 7 would be improved significantly. At the same time, the localization of faulty instances (rather than faulty components) would be more difficult. Even this form of coarse microreboot provided 1-2 orders of magnitude improvement over existing recovery approaches, hence reducing the motivation to pursue finer granularities.

4. Release all resources associated with the microrebooted component: locks, external resources like DB connections, in-progress transactions, etc. Note that only the resources allocated via the application server can be released. Given that Java does not offer explicit release of resources, the application server may choose to invoke the system-wide garbage collector after having nulled all references to resources.

   The only resource we do not discard on a microreboot is the component's classloader. JBoss uses a separate class loader for each EJB to provide appropriate sandboxing between components; when a caller invokes an EJB method, the caller's thread switches to the EJB's classloader. A Java class' identity is determined both by its name and the classloader responsible for loading it; discarding an EJB's classloader upon microreboot would unnecessarily complicate the update of internal references to the microrebooted component. Preserving the classloader does not violate any of the sandboxing properties. Keeping the classloader active does not reinitialize EJB static variables upon microreboot, but this is acceptable, since J2EE strongly discourages the use of mutable static variables anyway, as this would prevent transparent replication of EJBs in clusters.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

5. Destroying the container marks the point at which the component is entirely removed from the system. Calls to this component will now fail during the JNDI name-lookup process, because there is no component registered under the desired name; the caller's container again handles this exception transparently. As an optimization, we modified the JNDI registry to maintain a mapping for the microrebooting component, but mark it as temporarily unavailable; a suitable exception is then issued to whoever performs a lookup on that name.

6. Once the container has been removed from the system, the application server discards any metadata it maintained on behalf of that component. In the same way an OS kernel does for processes, JBoss maintains for each active EJB a rich set of metadata: the Java class implementing its functionality, the type of EJB (session, entity, etc.), whether the bean requires transactional support, along with references to other beans that this EJB might call and references to the resources required by the EJB, accounting information, security credentials, etc.

7. The worker thread pool needs to be replenished, because a number of threads have been killed off in step 2.

8. A new container is created and a new instance of the component is instantiated inside this container.

9. The `start()` method is a standard part of the EJB and WAR interface, as per the J2EE specification [Sun]. This method instructs the component to initialize itself. Note that the "reject incoming calls" flag is no longer set in the newly-created container, so both new and retried calls can come in, once the `start()` completes. We have not implemented a backoff mechanism to avoid overloading the newly instantiated component, but this could easily be added to the application server.

Since all persistent and session state is preserved in specialized state stores, behind narrow APIs, this microreboot does not lead to state corruption; the state updates ensure that consistency is preserved. As will be shown in chapter 7, this form of microreboot fixes many problems, such as EJB-private variables being corrupted, EJB-caused memory leaks, or the inability of one EJB to call another because its reference to the callee has become stale.

The time to reintegrate a microrebooted component is determined by the amount of initialization it performs at startup and the time it takes for other components to recognize the newly-instantiated EJB. Initialization (the `start()` method) dominates reintegration time; in our prototype it takes

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

on the order of hundreds of milliseconds, but varies considerably by component, as will be seen in table 7.2. The time required to destroy and re-establish EJB metadata in the application server is negligible. Making the EJB known to other components happens through the JNDI naming service described earlier; this level of indirection ensures immediate reintegration once the component is initialized.

Our implementation of microreboots does not scrub application-global data maintained by the application server, such as the JDBC connection pool and various other caches. Microreboots also generally cannot recover from problems occurring at layers below the application, such as the application server or the JVM. In all these cases, a full server restart may be required.

6.3.2 EBid: A crash-only application

Although many companies use JBoss to run their production applications, we found them unwilling to share their applications with us. Instead, we converted Rice University's RUBiS [CMZ02] (a J2EE/Web-based auction system that mimics the functionality of the popular eBay online auction service) into eBid – a crash-only version of RUBiS with additional functionality.

eBid maintains user accounts, allows bidding on, selling, and buying of items, has item search facilities, customized information summary screens, user feedback pages, etc. It distinguishes three kinds of users: visitor, buyer, and seller, with buyer and seller sessions requiring login. A buyer can bid on items and consult a summary of their current bids, rating and comments left by other users. Seller sessions require a "fee" before a user is allowed to put up an item for sale. The seller can specify a reserve (minimum) price for an item. eBid contains 582 Java files and about 26K lines of code; it uses MySQL for the database back-end and stores 7 tables. In our configuration, eBid has 132,000 items for sale, distributed among eBay's 40 categories and 62 regions. There are 1,500,000 entries in the bids table (i.e., an average of 11 bids/item). The users table has 10,000 entries.

6.3.3 State segregation in EBid

Like most e-commerce applications, eBid has long-term data, session data, and static presentation data. We keep these categories of state in a database, dedicated session state store, and an Ext3FS filesystem (optionally mounted read-only), respectively.

eBid presents a mixed object-oriented/procedural design, consistent with best practices for building scalable J2EE applications [CMZ02]. eBid uses entity EJBs and stateless session EJBs.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

The entity EJBs implement the persistent application objects, in the traditional object-oriented programming sense, with each instance's state mapped to a row in a database table. Stateless session EJBs are used to perform higher level operations on entity EJBs: each end-user operation is implemented by a stateless session EJB interacting with several entity EJBs. For example, there is a "place bid on item X" EJB that performs operations on three entity EJBs (User, Item, and Bid).

Persistent state in eBid consists of user account information, item information, bid/buy/sell activity, etc. and is maintained in a MySQL database through 9 entity EJBs: IDManager, User, Item, Bid, Buy, Category, OldItem, Region, and UserFeedback. MySQL is crash-safe and recovers fast for our data sets. Each entity bean uses container-managed persistence. If an EJB is involved in any transactions at the time of a microreboot, they are all automatically aborted by the container and rolled back by the database.

Session state in eBid takes the form of items that a user selects for buying/selling/bidding, her userID, etc. Users are identified using HTTP cookies. In our prototype, we keep session state outside the application, in a dedicated session state repository, consistent with crash-only design.

6.3.4 Session state storage

As we stated earlier, management of session state varies across implementations of J2EE application servers. JBoss offers two options: (a) individual EJB's can manage their own session state by explicitly updating an external state store; or (b) JBoss can transparently manage session state, which it does by keeping it in RAM with no replication or backup. In eBid, we use option (a), which means all beans' session state will survive microreboots of the beans themselves. The session state could be stored in a transactional database, but this would impose significant burdens on the application both in terms of using the SQL interface and in terms of performance.

We created two options for eBid's session state storage needs. First, we modified SSM [LKF04], a clustered session state store with a hashtable API. SSM maintains its state on separate machines; isolated by physical barriers, it provides access to session state, and survives application-level microreboots, JVM-level restarts, as well as node reboots. The session storage model is based on leases, so orphaned session state is garbage-collected automatically. Second, we built FastS, an in-memory repository inside JBoss's Web server. The API is identical to SSM; it illustrates how session state can be segregated from the application, yet still be kept within the same JVM. Isolated behind compiler-enforced barriers, FastS provides access to session objects much faster than SSM, but only survives application-level microrebooting – a full restart of the JVM or the node will discard all session state.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Both SSM and FastS are crash-only and guarantee atomic, consistent updates to the session state objects (i.e., the `put()` method). The session state abstraction takes advantage of the typical session state workload (objects are read/written by a single user in a serial fashion, and state has to persist only for the duration of a user session) in order to relax isolation and durability constraints in the implementation.

A major challenge when designing a state store isn't that much how to build it, but rather what abstractions to offer programmers, so as to make it easier for them to separate the state into suitable categories. Figure 6.4 shows stylized sample eBid code that uses the session state interface common to SSM and FastS, as a way to illustrate the abstraction provided for session state.

```
obj = new SessionObject;

// initialize session state

cookie = FastS.put( obj, lifetime );
obj = null;

// send cookie to client
// do other work
// receive cookie from client
obj = FastS.get( cookie );

// read/write session object

newCookie = FastS.put( obj, lifetime );

// send new cookie to client
```

Figure 6.4: Sample code using the SSM/FastS session state interface.

In response to a `put()` operation, the session state store returns a lookup key in the form of a cookie. Every time a `put()` is performed, a new key is generated and returned to the caller; this key uniquely identifies that version of the session state object. The key can conveniently be wrapped in an HTTP cookie and sent to the client Web browser, thus relieving the application from the need of maintaining any references to session state. The next time the client accesses the application, it sends the cookie, which provides the key needed to retrieve the user's session state (as illustrated in the sample code). This offers complete decoupling between client, application, and session state.

On each `put()`, the application can specify a *lifetime*, as an optional parameter. This instructs the session state store for how long that object has to be persisted; in the case of most e-commerce applications, the lifetime is on the order of 15-20 minutes. If *lifetime* is not specified, a system

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

default is used. Once *lifetime* expires, the state store will discard that version of the object; as a result, if session state is not updated for *lifetime* minutes, all versions will have expired and a new state object has to be created. The application does not have any means to explicitly delete session objects in FastS or SSM.

6.3.5 Recovery manager

The recovery manager is an entity external to the execution platform. It receives failure notifications and automatically attempts recovery; it only involves human operators when the recovery policy requires it to do so (e.g., when repeated automated recovery attempts are unsuccessful). The recovery manager performs recovery by sending microreboot requests to the microreboot facility.

Running the recovery manager outside the monitored system allows for independent failure and recovery of the recovery manager itself. To ensure the recovery manager is always running, there can be a mutual supervision relationship between the application platform and the recovery manager, that allows either one to detect the others' failure and restart it. The recover manager does extensive logging of its actions, so that failures can be debugged a posteriori, hence removing the diagnosis step from the critical path of recovery.

When the recovery manager receives a failure notification from the monitoring facility, it is responsible for making recovery decisions. To aid in these decisions, the manager maintains a dynamic view of the system that captures the currently known paths along which faults can propagate.

The system view is captured in a failure propagation map, as defined in section 5.2.1 – a graph that has application components as nodes and direct fault-propagation paths as edges. Given that the recovery manager has no a priori knowledge of the layout of the application or system it is supposed to manage, nor of what components form the system or how they interact, the recovery manager uses failure reports to infer the f-map.

There are a variety of techniques to build such an f-map, including static source code analysis or human-generated dependency graphs. We developed an application-generic tool, called AFPI (see appendix A). AFPI has two phases: in the (invasive) staging phase, the recovery manager actively performs both single-point and correlated fault injections, observes the system's reaction to the faults, and builds the "first draft" of the f-map; in the (non-invasive) production phase, the system passively observes fault propagation when such faults occur during normal operations, and uses this information to refine and evolve the f-map on an ongoing basis. In both phases, the monitors report to the recovery manager the path taken by faults through the system, and the manager adds the corresponding edges to the f-map. If components are added or removed from the system for upgrade

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

or reconfiguration reasons, the recovery manager is notified and automatically removes/adds the corresponding nodes from/to the f-map. The passive observation phase works fine even without the initial active phase, but can take much longer to converge onto a correct representation of the failure dependencies.

We built a recovery manager that performs simple failure detection and localization (described in more detail in section 6.4) and recovers the application by microrebooting EJBs, the WAR, or all of eBid; restarting the JVM that runs JBoss (and thus eBid as well); or rebooting the operating system. The recovery manager listens on a UDP port for failure reports coming in from the monitors; these reports take the form of a failure signature $f = <$*symptom, faulty URLs*$>$, containing the type of failure observed and the particular URLs that are not working as expected.

The recovery manager uses a recursive recovery policy based on the principle of trying the cheapest recovery first. If the cheapest recovery is not effective, the recovery manager reboots progressively larger subsets of components. Thus, it first microreboots EJBs, then eBid's WAR, then the entire eBid application, then the JVM running the JBoss application server, and finally reboots the OS; if none of these actions cure the failure symptoms, a human administrator is notified. In order to avoid endless cycles of rebooting, the recovery manager also notifies a human whenever it notices recurring failure patterns. The recovery action per se is performed by remotely invoking JBoss's `microreboot()` method (for EJB, WAR, and eBid) or by executing commands, such as `kill -9`, over `ssh` (for JBoss and node-level reboot).

6.4 Failure Detection and Localization

An important part of fast recovery is fast detection and accurate localization. Particularly in the case of microrebooting, good localization is very important.

Downtime for an incident consists of the time it takes for the failure to be detected by a monitor (T_{det}), the time to diagnose the problem (T_{diag}), and the time to recover (T_{rec}):

$$T_{\mathsf{down}} = T_{\mathsf{det}} + T_{\mathsf{diag}} + T_{\mathsf{rec}}$$

While detection and diagnosis is out of scope for this dissertation, we did explore the topic as much as was needed to build a realistic prototype.

Since the only recovery method employed is microreboot, which is application-generic, we don't need diagnosis; in the case of full reboot, detection of a reboot-curable failure is sufficient;

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

in the case of microreboot, detection *and localization* is sufficient. Diagnosis is different from localization (i.e., identification of the faulty component); we argue that a good approach to recovery is to do sufficient localization to perform the recovery, and then do all diagnosis out of the critical recovery path (e.g., offline based on logs). As a result, T_{down}'= T_{det} +T_{rec} and T_{down}'<<T_{down}, because T_{diag} often involves humans.

6.4.1 Failure detection

There is a plethora of commercial solutions available for system monitoring, most of which do not do component-level monitoring; this is understandable, since fine-grained localization of failures is pointless if one does not have the ability to do microrecovery (it only becomes useful for debugging, profiling, etc.)

To enable fully-automatic recovery, we implemented failure detection in the client emulator (section 6.5) and placed primitive localization facilities in the external recovery manager. While real end users' Web browsers certainly do not report failures to the Internet services they use, our client-side detection mimics WAN services that deploy "client-like" end-to-end monitors around the Internet to detect a service's user-visible failures [Key]. Such a setup allows our measurements to focus on the recovery aspects of our prototype, rather than the orthogonal problem of detection and diagnosis.

We implemented two fault detectors. The first one is simple and fast: if a client encounters a network-level error (e.g., cannot connect to server) or an HTTP 4xx or 5xx error, then it flags the response as faulty. If no such errors occur, the received HTML is searched for keywords indicative of failure (e.g., "exception," "failed," "error"). Finally, the detection of an application-specific problem can also mark the response as faulty (such problems include being prompted to log in when already logged in, encountering negative item IDs in the reply HTML, etc.)

The second fault detector submits each request to the application instance we are injecting faults into, as well as (in parallel) to a separate, known-good instance on another machine. It then compares the result of the former to the "truth" provided by the latter, flagging any differences as failures. This detector is the only one able to identify complex failures, such as the surreptitious corruption of the dollar amount in a bid. Certain additional checks were required to account for timing-related nondeterminism.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

6.4.2 Score-based fault localization

Using static analysis, we derived a mapping from each eBid URL prefix to a path/sequence of calls between servlets and EJBs. The recovery manager maintains for each component in the system a score, which gets incremented every time the component is in the path originating at a failed URL. The recovery manager decides what and when to (micro)reboot based on hand-tuned thresholds. Accurate or sophisticated failure detection was not the topic of this work; our simple approach to diagnosis often yields false positives, but part of our goal is to show that even the mistakes resulting from simple or "sloppy" diagnosis are tolerable because of the very low cost of microreboots – more on this topic in chapter 8.

6.4.3 Inferring application structure

We obtained a description of the failure dependencies between eBid's components using automated fault-propagation inference (described in appendix A); the resulting fault propagation map is shown in figure 6.5. Shaded components are EJBs, clear boxes indicate servlets. We found AFPI to be more precise than just statically inspecting the application's deployment descriptors; static inspection of the source code, however, could work quite well, but would be manual. This f-map is used by the recovery manager to compute recovery groups.

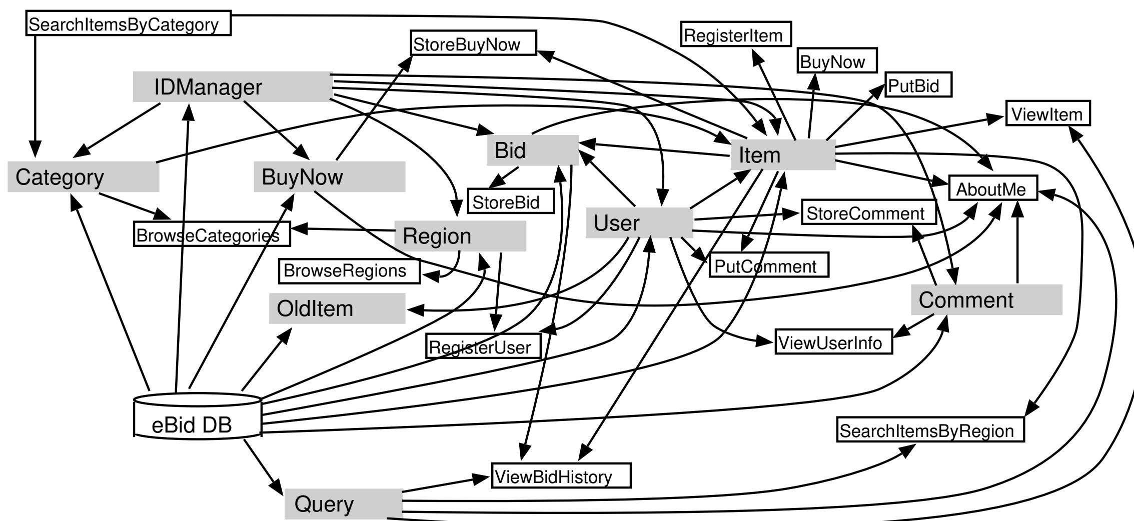

Figure 6.5: An f-map for eBid, obtained with AFPI.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

6.5 Client Emulator

We wrote a client emulator using some of the logic in the load generator shipped with RUBiS. Human clients are modeled using a Markov chain with 25 states corresponding to the various end user operations possible in eBid, such as *Login*, *BuyNow*, or *AboutMe*; transitioning to a state causes the emulated client to issue a corresponding HTTP request. In addition to the application-specific states, we also have two states corresponding to the user hitting the back button (*Back*) and spontaneously deciding to end his/her session and abandon the site (*End*).

Transition probabilities are stored in a table T. The client emulator uses this table to automatically navigate the Web site; when in a given state s, it will choose the next state t randomly, with probability $T(s,t)$; it then constructs the URL for this state and "clicks" on it. Inbetween successive "clicks," emulated clients have independent think times based on an exponential random distribution with a mean of 7 seconds and a maximum of 70 seconds, as in the TPC-W Web server benchmark [Smi02].

We populated table T with transition probabilities representative of online auction users; the resulting workload, shown in Table 6.1, mimics the real workload seen by a major Internet auction site [EBa04].

User operation results mostly in...	**% of all requests**
Read-only DB access (e.g., browse a category)	32%
Initialization/deletion of session state (e.g., login)	23%
Exclusively static HTML content (e.g., home page)	12%
Search (e.g., search for items by name)	12%
Session state updates (e.g., select item for bid)	11%
Database updates (e.g., leave seller feedback)	10%

Table 6.1: Client workload used in evaluating microreboot-based recovery.

We classify responses from the server as correct or incorrect (see section 6.4). Our chosen workload covers all possible eBid operations; experimentally we have determined that, in runs lasting a few minutes with 20 clients or more, we routinely exercised all components.

6.6 Action-Weighted Throughput

In choosing a metric for evaluating microreboot-based recovery, we wanted to find a way to capture the impact such recovery has on end users. A simple approach to measuring the effect of downtime

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

on end users would be to measure goodput (number of requests completed successfully) under partial-failure conditions, averaged across all clients. This is usually how performability [Mey80] is measured.

Unfortunately, the simple goodput metric fails to distinguish between potentially long-running DB-touching operations and simple, fast, browse-only operations. A typical interaction of a client with the Web site proceeds as follows: client goes to the homepage, browses around for a while performing different site actions (e.g., searching), and then decides to do something that touches the persistent-state database (e.g., place a bid, leave a comment for a user, update his/her profile, etc.). The DB-touching operation(s) usually require the user to have logged in. Interactions preceding the persistent-state update are just precursors to the real action, making the DB update a sort of "commit point" from the point of view of users' behavior; thus, these interactions serve no purpose (with respect to the proposed metric) in the absence of a successful commit point.

The surprising result of inducing failures during long-running DB operations is that the goodput actually goes *up* in the presence of failure, because the user no longer waits for a long-running operation – it fails right away and the client emulator moves on to the next (non-DB-touching) operation. In other words, the simple goodput metric would fail to capture that some operations are more "valuable" than others, and executing many "simple" operations does not necessarily compensate for failing to execute a few long-running ones.

We therefore devised a new metric that accurately reflects *end-user perceived* availability rather than some arbitrary notion of *true* availability.

In action-weighted throughput (T_{aw}), we view a user *session* as beginning with a login operation and ending with an explicit logout or abandonment of the site. In our model, if something goes wrong during a session, the user will try to logout and log back in via the homepage, so we define a session as the sequence of URLs bracketed by accesses to the homepage.

Each session consists of a sequence of user *actions*. In our model, each user action is a sequence of *operations* (HTTP requests) that culminates with a "commit point": an operation that must succeed for that user action to be considered successful as a whole. For example, the last operation in the action of placing a bid results in committing that bid to the database; similarly, we assume that browsing activity has the purpose of performing some action that carries the meaning of a commit point.

An action succeeds or fails atomically: if all operations within the action succeed, they count toward action-weighted goodput ("good T_{aw}"); if an operation fails, all operations in the corresponding action are marked failed, counting toward action-weighted badput ("bad T_{aw}"). Unlike

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

simple throughput, T_{aw} accounts for the fact that both long-running and short-running operations must succeed for a user to be happy with the service. T_{aw} also captures the fact that, when an action with many operations succeeds, it generally means the user did more work than in a short action. Figure 7.1 gives an example of how we use T_{aw} to compare recovery by microreboot to recovery by JVM restart.

Commit points are those user actions that indicate completion of useful work on the part of the end-user. In our auction application, we identified 9 commit points: the actions that correspond to database updates (registering a new user, auctioning a new item, making a bid, performing a "buy now," leaving user feedback) as well as actions that indicate completion of useful work even without a database update (viewing account information, logging out, spontaneously deciding to abandon the site, and clicking to the homepage after a sequence of browse operations).

6.7 Chapter Summary

In this chapter we described the details of a J2EE-based testbed for microreboot-based recovery. Our choices were motivated by the synergy that exists between J2EE and the principles of crash-only design. After introducing the J2EE framework, we presented the extensions we made to JBoss and other software used in our experiments, described our failure detection and localization mechanism, the client emulator, and finally the action-weighted throughput metric used for the evaluation.

This testbed was hosted on 3GHz Pentium machines with 1GB RAM for Web and JBoss tier nodes; databases were hosted on Athlon 2600xp+ machines with 1.5 GB of RAM and 7200rpm 120GB disks; emulated clients ran on a variety of multiprocessor machines. All machines were interconnected by a 100 Mbps Ethernet switch and ran Linux kernel 2.6.5 with Sun Java 1.4.1 and Sun J2EE 1.3.1.

In the next chapter we describe the results obtained using this testbed.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 7

Evaluation of the Microreboot Recovery Mechanism

We used our prototype to answer four questions about a microreboot-based approach to recovery in J2EE enterprise applications:

- Is microrebooting effective in recovering from failures in J2EE systems? In particular, how does it compare to a JVM process restart?
- Is there any benefit to using component-level microreboot instead of JVM restarts? If yes, how large is this benefit?
- Is microrebooting useful in clusters of J2EE application servers? If yes, how and to what extent?
- Does a crash-only J2EE system incur any performance overhead? If yes, is this significant?

We find that microrebooting is able to recover from a substantial majority of the failures that could be cured by a process restart (section 7.1), but does so 50x faster and with a 98% reduction in user-perceived downtime (section 7.2). This results from the fact that microreboots recover faster, induce less functional disruption, and preserve more critical state. In clusters of application servers, microrebooting plays an important role in recovery performance, by reducing both the number of failed requests as well as the number of failed-over requests; should failure be accompanied by a load spike (as it often is), response time is better preserved when microreboot-based recovery is employed (section 7.3). Finally, any performance degradation introduced by a crash-only design is insignificant when comparing to top commercial systems (section 7.4).

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

7.1 Recovery Effectiveness

Having built a realistic testbed and generated realistic workloads, we wanted to inject realistic faults in our prototype and observe its ability to recover from the ensuing failures.

Despite J2EE's popularity, we were unable to find any published systematic studies of faults occurring in production J2EE systems. Therefore, we interviewed 10 professionals who work with enterprise applications or application servers in variety of industry sectors, and asked them about the failures they experience[1]. In deciding what faults to inject in our prototype, we relied on these interviews as well as advice from colleagues in industry, who operate systems such as ours [Cho03, CJ02, Lev03, Mit04, Pal02, Rei04].

We found that production J2EE systems are most frequently plagued by deadlocked threads, leak-induced resource exhaustion, bug-induced corruption of volatile metadata, and various Java exceptions that are handled incorrectly. These results are not surprising: First, J2EE systems are highly threaded and the interactions between these threads can often lead to deadlocks. Second, the workloads faced by Web-connected applications are typically very large, so even the smallest resource leak will quickly grow into a system-wide resource shortage that causes the application to fail. Finally, given the complexity and heterogeneity of these large-scale environments, it is difficult for application code to correctly handle all the possible exception conditions that arise in practice.

We added hooks in JBoss for injecting artificial deadlocks, infinite loops, memory leaks, JVM memory exhaustion outside the application, transient Java exceptions to stress eBid's exception handling code, and corruption of various data structures. In addition to these hooks, we also used FIG [BST02] and FAUmachine [BS01] to inject low-level faults underneath the JVM layer.

eBid, being a crash-only application, has relatively little volatile state that is subject to loss or corruption – much of the application state is kept in FastS / SSM. We can, however, inject faults in the data handling code, such as the code that generates application-specific primary keys for identifying rows in the DB corresponding to entity bean instances. We also corrupt class attributes of the stateless session beans. In addition to application data, we corrupt metadata maintained by the application server, but accessible to eBid code: the JNDI repository, that maps EJB names to their containers, and the transaction method map stored in each entity EJB's container. Finally, we corrupt data inside the session state stores (via bit flips) and in the database (by manually altering table contents).

We performed three types of data corruption on various fields: (a) set a value to *null*, which

[1] All these interviews were conditional upon not disclosing the sources.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

generally elicits a `NullPointerException` upon access; (b) set an *invalid* value, i.e., a non-null value that type-checks but is invalid from the application's point of view, such as a userID larger than the maximum userID; and (c) set to a *wrong* value, which is valid from the application's point of view, but incorrect, such as swapping IDs between two users.

After injecting a fault, we used the recursive policy described earlier to recover the application. In reporting the results, we differentiate between *resuscitation* (restoring the system to a point from which it can resume the serving of requests for all users, without necessarily having fixed the resulting database corruption) and *recovery* (bringing the system to a state where it functions with a 100% correct database). For example, contrary to popular belief, financial institutions typically aim for resuscitation, applying compensating transactions at the end of the business day to repair database inconsistencies [Rei04]. In the rightmost column, we indicate whether additional manual database repair actions were required to achieve correct recovery after resuscitation.

In Table 7.1 we show the worst outcome among several experiments for each type of injected fault. There are several cases (e.g., bit flips) in which complete recovery was achieved most of the times, but on rare occasions only resuscitation could be achieved; in such cases, manual intervention was required, and the table lists "resuscitation" as a result.

For this experiment we relied on our comparison-based failure detector to determine whether a recovery action had been successful or not; when failures (caused by a difference in the responses of the good vs. faulty application server instances) were encountered subsequent to the recovery, we escalated recovery to the next level in the policy.

Aside from EJB, JBoss, and operating system reboots, some faults required microrebooting eBid's Web component (WAR). In two cases no resuscitation was needed, because the injected fault is "naturally" expunged from the system after the first call fails. In the case of recovering persistent data, this is either done automatically (transaction rollback), or, in the case of injecting *wrong* data, manual reconstruction of the data in the DB is often required (indicated by a "resuscitated" value in the last column).

In our broader experience, we have found few failures from which whole-system restart recovers but microreboots do not. Nevertheless, such failures exist; for example, we observed that under high load, the JVM running the application server would sometimes run out of file descriptors, or encounter an internal error, requiring a process restart of the JVM. We have also encountered a resource leak involving serialized objects sent over a socket: the object does not get garbage collected even when our references to it are gone, and eventually the leaks require a JVM restart. Finally, on our version of Linux, we also encountered on occasion a kernel bug in the swapping

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Injected Fault	**Type**	**Reboot level**	**Result**
Deadlock		EJB	recovered
Infinite loop		EJB	recovered
Application memory leak		EJB	recovered
Transient exception		EJB	recovered
Corrupt primary keys	set null	EJB	recovered
	invalid	EJB	recovered
	wrong	EJB	resuscitated
Corrupt JNDI entries	set null	EJB	recovered
	invalid	EJB	recovered
	wrong	EJB	recovered
Corrupt transaction method map	set null	EJB	recovered
	invalid	EJB	recovered
	wrong	EJB	resuscitated
Corrupt stateless session EJB attributes	set null	unnecessary	recovered
	invalid	unnecessary	recovered
	wrong	EJB+WAR	resuscitated
Corrupt data inside FastS	set null	WAR	recovered
	invalid	WAR	recovered
	wrong	WAR	resuscitated
Corrupt data inside SSM	corruption detected via checksum; bad object automatically discarded		
Corrupt data inside MySQL	database table repair needed		
Memory leak outside application	intra-JVM	JVM/JBoss	recovered
	extra-JVM	OS kernel	recovered
Bit flips in process memory		JVM/JBoss	resuscitated
Bit flips in process registers		JVM/JBoss	resuscitated
Bad system call return values		JVM/JBoss	recovered

Table 7.1: Recovery from injected faults: worst-case scenarios.

code which would manifest under high memory utilization conditions; in such cases, any memory allocation (specifically, any call to the `brk` system call) hangs, and a full system restart is necessary.

The results indicate that EJB-level or WAR-level microrebooting in our J2EE prototype is effective in recovering from the majority of failure modes seen in today's production J2EE systems (first 19 rows of Table 7.1). Microrebooting is ineffective against other types of failures (last 7 rows), where coarser grained reboots or manual repair are required. Fortunately, these failures constitute an insignificant fraction of failures in real J2EE systems. While certain faults (e.g., JNDI corruption) could certainly be cured with non-reboot approaches, we consider the reboot-based approach simpler, quicker, and more reliable. In the cases where manual actions were required to restore

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

service correctness, a JVM restart presented no benefits over a component-level microreboot.

While rebooting is a common way to recover middleware in the real world, for the rest of this paper we compare EJB-level microrebooting to JVM process restart (which restarts JBoss and, implicitly, eBid) rather than full OS/node-level reboots. In doing so, we are conservative with respect to the results of our experiments; if we compared against OS/node-level rebooting, our results would be more favorable, since a machine reboot takes several times (5x in our case) longer than a process restart.

7.2 Recovery Efficiency

With respect to availability, Internet service operators care mostly about how many user requests their system turns away during downtime, not the absolute value of downtime (e.g., 1 minute of downtime can cost $0 in the middle of the night, whereas that same minute of downtime can cost $50,000 [Kem98] during peak time). We therefore evaluate the efficiency of microrebooting with respect to the end-user-aware action-weighted throughput metric (T_{aw}).

We injected faults in our prototype and then allowed the recovery manager to recover the system automatically in two ways: by restarting the JVM process running JBoss, or by microrebooting one or more EJBs, respectively; we compare the results of these two forms of recovery. We ran the same workload and faultload for both the process restart and microreboot experiments. We deem recovery successful when end users do not experience any more failures after recovery, as determined by our application-specific fault detector.

Figure 7.1 shows the results of injecting three different faults every 10 minutes. Each sample point represents the number of successful (failed) requests observed during the corresponding second. While we did perform this experiment for many of eBid's components, we show here only a conservative sample. We injected the following faults:

1. At $t = 10$ minutes, we corrupted the transaction method map for *EntityGroup*, the EJB recovery group that takes the longest to recover (see table 7.2).

2. At $t = 20$ minutes, we corrupted the JNDI entry for *RegisterNewUser*, the EJB with the slowest individual recovery time.

3. At $t = 30$ minutes, we injected an exception in *BrowseCategories*, the entry point for all browsing on the eBid Web site and, thus, the most-frequently called EJB in our workload. An

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

outage in *BrowseCategories* would therefore have the most immediately visible impact of all components on the user population.

Session state is stored in FastS. We ran a load of 500 concurrent clients connected to one application server node; for our specific setup, this lead to a CPU load average of 0.7, which is similar to that seen in deployed Internet systems [Mes04b, Duv04]. Unless otherwise noted, we use 500 concurrent clients per node in each subsequent experiment.

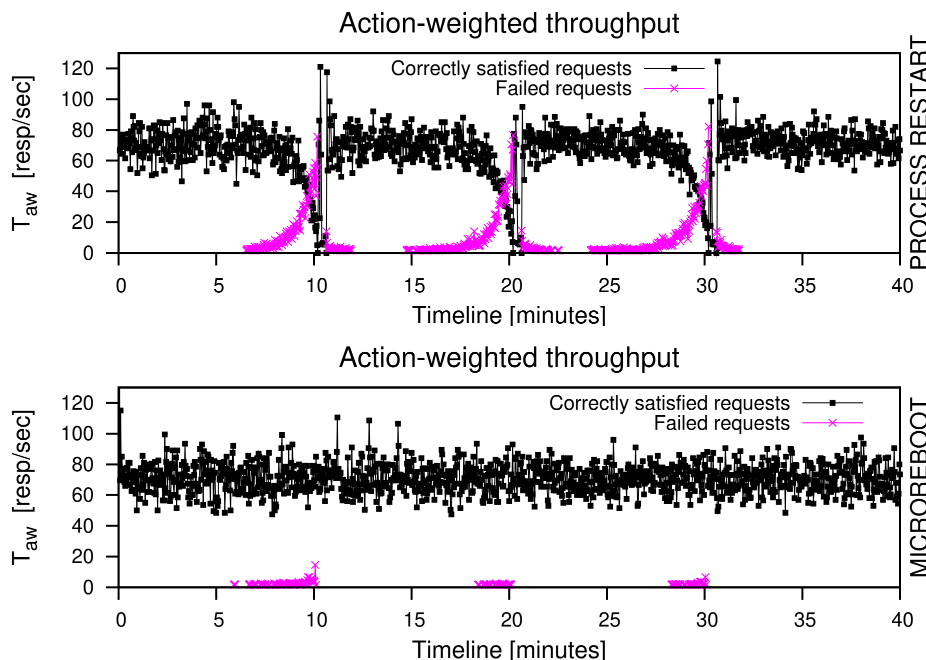

Figure 7.1: Impact of recovery on end users: process restart vs. microreboot.

In figure 7.1, recovering with a process restart (shown in the top graph) caused 11,752 requests to fail overall, corresponding to 3,101 user actions. When microrebooting, only 233 requests (34 actions) failed. In the latter case, three recovery actions were undertaken, that required 7 microreboots, due to the dependencies between components[2]. The average is 3,917 failed requests (1,034 actions) per process-restart-based recovery action, and 78 failed requests (11 actions) per microreboot-based recovery action.

Using microreboots instead of JVM restarts reduced the number of failed requests by 98% in this conservative experiment; this corresponds to a factor of 50x reduction in failed end-user

[2]In fact, 10 microreboots occurred, because of lack of precision in our fault localization algorithm (6 microreboot recovery actions instead of 3); despite this, the results are better than the case of process restarts. See section 8.1 for a more detailed discussion of this aspect.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

requests. Visually, the impact of a failure and recovery event can be estimated by the area of the corresponding dip in good T_{aw}, with larger dips indicating higher service disruption. The area of a T_{aw} dip is determined by its width (i.e., time to recover) and depth (i.e., the throughput of requests turned away during recovery). We now analyze these factors in isolation.

7.2.1 Faster recovery

The wider the dip in T_{aw}, the more requests arrive during recovery; since these requests fail, they cause the corresponding user actions to fail, thus retroactively marking the actions' requests as failed. This explains why "bad T_{aw}" in figure 7.1 is retroactive relative to the injection point.

We measured recovery time at various granularities and summarize the results in Table 7.2. In the two right columns we break down recovery time into how long the target takes to crash (be forcefully shut down) and how long it takes to reinitialize. EJBs recover an order of magnitude faster than JVM restart, which explains why the width of the good T_{aw} dip in the microreboot case of figure 7.1 is negligible. EJBs with a * superscript are entity EJBs, while the rest are stateless session EJBs. Averages are computed across 10 trials per component, on a single-node system under sustained load from 500 concurrent clients. Recovery for individual EJBs ranges from 411-601 msec.

Some EJBs have inter-dependencies, captured in deployment descriptors, that require them to be microrebooted together. eBid has one such recovery group, *EntityGroup*, containing 5 entity EJBs: *Category*, *Region*, *User*, *Item*, and *Bid* – any time one of these EJBs requires recovery, we microreboot the entire *EntityGroup*. Restarting the entire eBid application is optimized to avoid restarting each individual EJB, which is why eBid takes less than the sum of all components to crash and start up. For the JVM crash, we use operating system-level `kill -9`.

Notice that all reboot-based recovery times are dominated by initialization. In the case of JVM-level restart, 56% of the time is spent initializing JBoss and its more than 70 services (transaction service takes 2 sec to initialize, embedded Web server 1.8 sec, JBoss's control & management service takes 1.2 sec, etc.). Most of the remaining 44% startup time is spent deploying and initializing eBid's EJBs and WAR. For each EJB, the deployer service verifies that the EJB object conforms to the EJB specification (e.g., has the required interfaces), then it allocates and initializes a container, sets up an object instance pool, sets up the security context, inserts an appropriate name-to-EJB mapping in JNDI, etc. Once initialization completes, the individual EJBs' `start()` methods are invoked. Removing an EJB from the system follows a reverse path.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Component name	Microreboot time (msec)	Crash (msec)	Reinit (msec)
AboutMe	551	9	542
Authenticate	491	12	479
BrowseCategories	411	11	400
BrowseRegions	416	15	401
BuyNow*	471	9	462
CommitBid	533	8	525
CommitBuyNow	471	9	462
CommitUserFeedback	531	9	522
DoBuyNow	427	10	417
EntityGroup*	825	36	789
IdentityManager*	461	10	451
LeaveUserFeedback	484	10	474
MakeBid	514	9	515
OldItem*	529	10	519
RegisterNewItem	447	13	434
RegisterNewUser	601	13	588
SearchItemsByCategory	442	14	428
SearchItemsByRegion	572	8	564
UserFeedback*	483	11	472
ViewBidHistory	507	11	496
ViewUserInfo	415	10	405
ViewItem	446	10	436
WAR (Web component)	1,028	71	957
Entire eBid application	7,699	33	7,666
JVM/JBoss process restart	19,083	≈ 0	19,083

Table 7.2: Average recovery times under load, at various granularities.

7.2.2 Less functional disruption

Figure 7.1 shows that good T_{aw} drops all the way to zero during a JVM restart, i.e., the system serves no requests during that time. In the case of microrebooting, though, the system continues serving requests while the faulty component is being recovered. We illustrate this effect in figure 7.2, graphing the availability of eBid's functionality as perceived by the emulated clients. We group all eBid end user operations into 4 functional groups (shown in different colors) – Bid/Buy/Sell, Browse/View, Search, and User Account operations – and zoom in on one of the recovery events of figure 7.1.

For each interval [t_1,t_2] along the horizontal axis, a solid bar indicates that all requests submitted

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

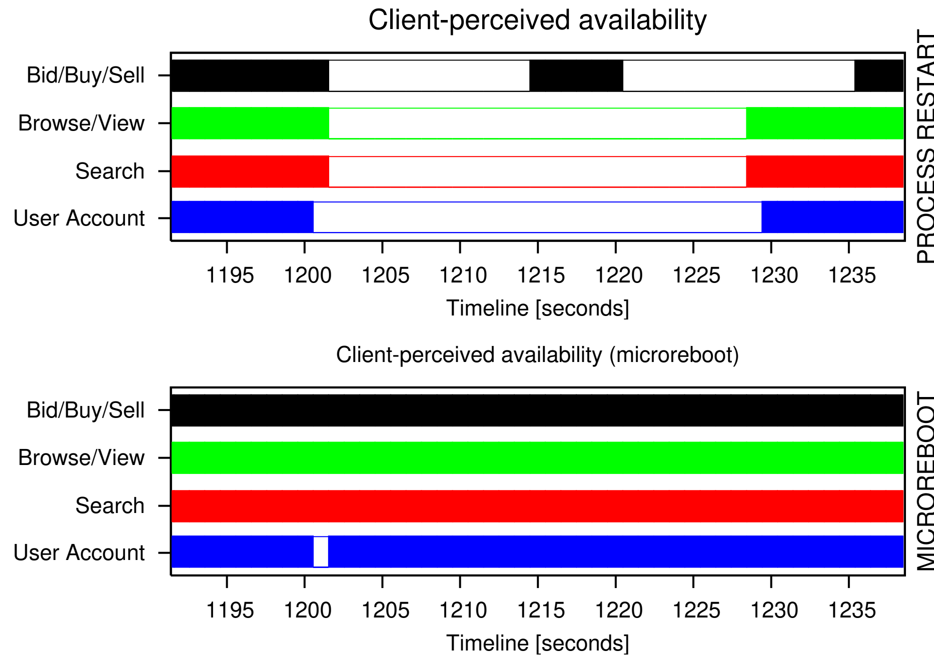

Figure 7.2: Functional disruption, as perceived by end users.

to the service during that interval in the corresponding category were correctly satisfied. A gap indicates that some request, whose processing spanned $[t_1,t_2]$ in time, eventually failed, conveying to the end user that the site was down.

While the faulty component is being recovered by microrebooting, all operations in other functional groups succeed. Even within the "User Account" group itself, many operations are served successfully during recovery (however, since *RegisterNewUser* requests fail, we show the entire group as unavailable). Fractional service degradation compounds the benefits of swift recovery, further increasing end user-perceived availability of the service.

7.2.3 Less lost work

In figure 7.1, a number of requests fail *after* JVM-level recovery has completed; this does not happen in the microreboot case. These failures are due to the session state having been lost during recovery (FastS does not survive JVM restarts). Had we used SSM instead of FastS, the JVM restart case would not have exhibited failed requests following recovery, and a fraction of the retroactively failed requests would have been successful, but the overall good T_{aw} would have been slightly lower, due to performance overhead. Using microreboots in the FastS case allowed the system to both preserve

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

session state across recovery and avoid cross-JVM access penalties.

7.2.4 Summary

In this section we evaluated the efficiency of microrebooting with respect to the end-user-aware action-weighted throughput metric T_{aw}. Results show that replacing JVM restarts in our J2EE application with microreboots reduces the number of failed requests by a factor of 50 in our experiments. This reduction is due to three effects: microreboots are faster than process restarts, microrebooting induces less user-visible functional disruption, and microrebooting preserves more user work than process restart.

7.3 Microrebooting in a Cluster

In a typical Internet cluster, the unit of recovery is a full node, which is small relative to the cluster as a whole. To learn whether microreboots can yield any benefit in such systems, we built a cluster of 8 independent application server nodes. Clusters of 2-4 J2EE servers are typical in enterprise settings, with high-end financial and telecom applications running on 10-24 nodes [Duv04]; a few gigantic services, like eBay's online auction service, run on pools of clusters totaling 2000 application servers [CZL+04].

Our testbed cluster consists of 8 application server nodes, 8 database nodes, and 8 Web server nodes. We distribute incoming load among nodes using a client-side load balancer *LB*. Under failure-free operation, *LB* distributes new incoming login requests evenly between the nodes and, for established sessions, *LB* implements session affinity (i.e., non-login requests are directed to the node on which the session was originally established). We inject a microreboot-recoverable fault from Table 7.1 in one of the server instances, say N_{bad}; the failure detectors notice failures and report them to the recovery manager. When *RM* decides to perform a recovery, it first notifies *LB*, which redirects requests bound for N_{bad} uniformly to the good nodes; once N_{bad} has recovered, *RM* notifies *LB*, and requests are again distributed as before the failure.

The general architecture of the testbed is shown in figure 7.3.

7.3.1 Conserving session state during failover

We first explored the configuration that is most likely to be found in today's systems: session state stored locally at each node; we use FastS. During failover, those requests that do not require session state, such as searching or browsing, will be successfully served by the good nodes; requests

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

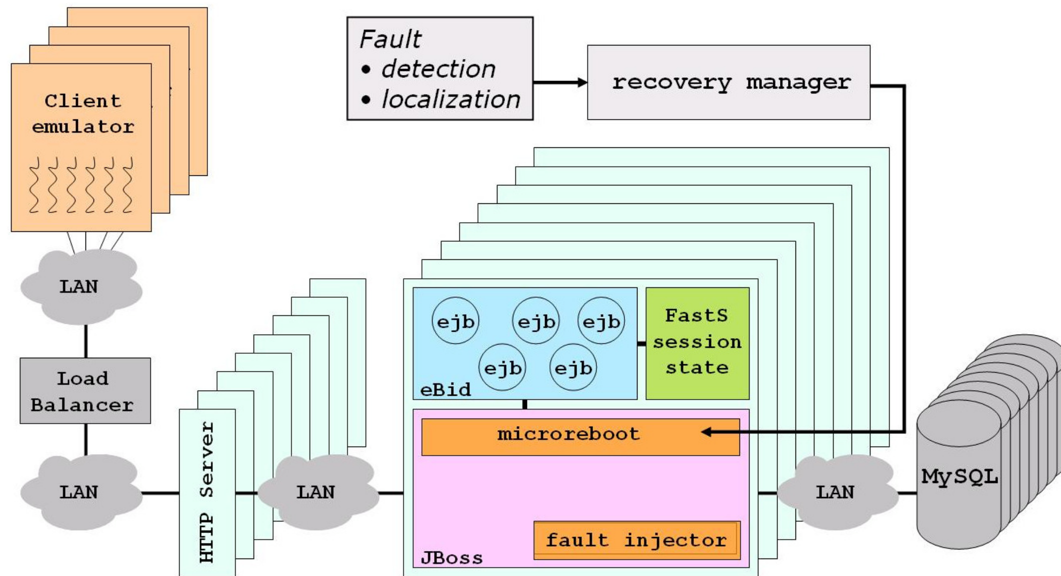

Figure 7.3: Conceptual architecture of the cluster testbed.

that require session state will fail. We injected a fault in the most-frequently called component (*BrowseCategories*) and ran the experiment in four different cluster configurations; the load was 500 clients/node.

The left graph in figure 7.4 shows the number of requests and failed-over sessions for the case of JVM restart and microreboot, respectively. When recovering N_{bad} with a JVM restart, the number of user requests that fail is dominated by the number of sessions that were established at the time of recovery on N_{bad}. In the case of EJB-level microrebooting, the number of failed requests is roughly proportional to the number of requests that were in flight at the time of recovery or were submitted during recovery. Thus, as the cluster grows, the number of failed user requests stays fairly constant. When recovering with JVM restart, on average 2,280 requests failed; in the case of microrebooting, 162 requests failed.

Although the relative benefit of microrebooting decreases as the number of cluster nodes increases (right graph in figure 7.4 – fraction of total user requests failed in our test's 10-minute interval, as a function of cluster size), recovering with a microreboot will always result in fewer failed requests than a JVM restart, regardless of cluster size or of how many clients each cluster node serves. Thus, it always improves availability. If a cluster aimed for the level of availability offered by today's telephone switches, then it would have to offer six nines of availability, which roughly means it must satisfy 99.9999% of requests it receives (i.e., fail at most 0.0001% of them).

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

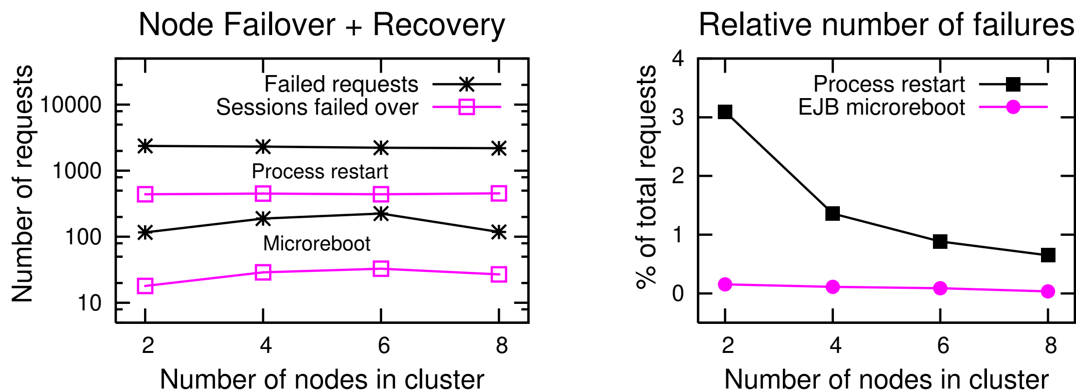

Figure 7.4: Failover under normal load.

Our 8-node cluster served 33.8×10^4 requests over the course of 10 minutes; extrapolated to a 24-node cluster of application servers, this implies 53.3×10^9 requests served in a year, of which a six-nines cluster can fail at most 53.3×10^3. If using JVM restarts, this number allows for 23 single-node failovers during the whole year; if using microreboots, as many as 329 failures would be permissible.

We repeated some of the above experiments using SSM. The availability of session state during recovery was no longer a problem, but the per-node load increased during recovery, because the good nodes had to (temporarily) handle the N_{bad}-bound requests. In addition to the increased load, the session state caches had to be populated from SSM with the session state of N_{bad}-bound sessions. These factors resulted in an increased response time that often exceeded 8 sec when using JVM restarts; microrebooting was sufficiently fast to make this effect unobservable. Overload situations are mitigated by overprovisioning the cluster, so we investigate below whether microrebooting can reduce the need for additional hardware.

7.3.2 Preserving response time during failover

We repeated the experiments from the previous section using FastS, but doubled the concurrent user population to 1000 clients/node. The load spike we model is very modest compared to what can occur in production systems (e.g., on 9-11, CNN.com faced a 20-fold surge in load, which caused their cluster to collapse under congestion [LeF01]). We also allow the system to stabilize at the higher load prior to injecting faults (for this reason, the experiment's time interval was increased to 13 minutes). JVM restarts are more disruptive than microreboots, so introducing a modest two-fold

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

change in load and inducing stability in initial conditions favors full process restarts far more than microreboots. Consequently, the microreboot results shown here are conservative.

Figure 7.5 shows that response time was preserved while recovering with microreboots, unlike when using JVM restarts. We show average response time per request, computed over 1-second intervals, in 4 different cluster configurations (2, 4, 6, 8 nodes). eBid uses FastS for storing session state, in both the JVM restart and microreboot case. Vertical scales of the four graphs differ, to enhance visibility of details.

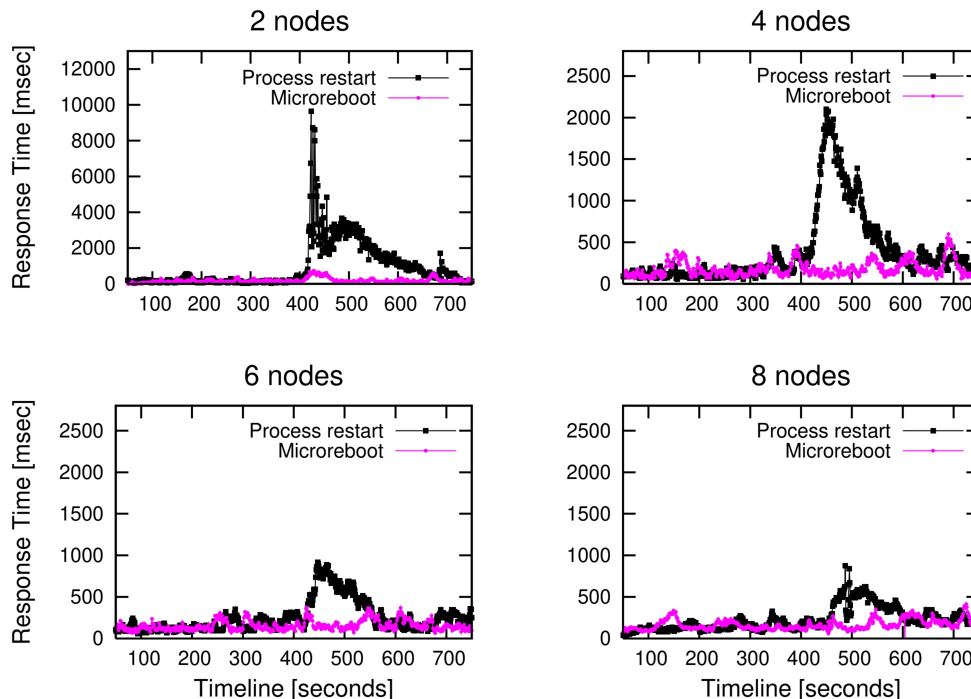

Figure 7.5: Failover under doubled load.

Stability of response time results in improved service to the end users. It is known that response times exceeding 8 seconds cause computer users to get distracted from the task they are pursuing and engage in others [Mil68, BBK00], making this a common threshold for Web site abandonment [Zon01]; not surprisingly, service level agreements at financial institutions often stipulate 8 seconds as a maximum acceptable response time [Mes04a]. We therefore measured how many requests exceeded this threshold during failover; Table 7.3 shows the corresponding results.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

# of nodes	**2**	**4**	**6**	**8**
Process restart	3,227	530	55	9
EJB microreboot	3	0	0	0

Table 7.3: Requests exceeding 8 sec during failover under doubled load.

7.3.3 Summary

Microreboots reduce the need for overprovisioning or sophisticated load balancing. We asked our colleagues in industry whether commercial application servers do admission control when overloaded, and were surprised to learn they currently do not [Mes04b, Duv04]. For this reason, cluster operators need to significantly overprovision their clusters and use complex load balancers, tuned by experts, in order to avert overload and oscillation problems. Microreboots are more successful at keeping response times below 8 seconds in our prototype.

The results of this section show that, by moving from process-restart-based recovery to microreboot-based recovery, cluster operators can expect to either

- provide higher levels of availability while keeping the same number of cluster nodes, or
- reduce the number of nodes while maintaining the same level of availability, which presents a cost and manageability benefit.

7.4 Performance Impact

In this section we measure the performance impact our modifications have on steady-state fault-free throughput and latency. We measure the impact of our microreboot-enabling modifications on the application server, by comparing original JBoss 3.2.1 (*JBoss*) to the microreboot-enabled variant ($JBoss_{\mu RB}$). We also measure the cost of externalizing session state into a remote state store by comparing eBid with FastS ($eBid_{FastS}$) to eBid with SSM ($eBid_{SSM}$).

It is not meaningful to compare the performance of eBid to that of original RUBiS (based on which we developed eBid), because the semantics of the applications are different. For example, RUBiS requires users to provide a username and password *each time* they perform an operation requiring authentication. In eBid, users log in once at the beginning of their session; they are subsequently identified based on the HTTP cookies they supply to the server on every access. We refer the reader to [CMZ02] for a detailed comparison of performance and scalability for various architectures in J2EE applications.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Table 7.4 summarizes the results. Throughput varies less than 2% between the various configurations, which is within the margin of error. Latency, however, increases by 70-90% when using SSM, because moving state between JBoss and a remote session state store requires the session object to be marshalled, sent over the network, then unmarshalled; this consumes more CPU than if the object were kept inside the JVM.

Configuration	**Throughput [req/sec]**	**Average Latency [msec]**
JBoss + $\text{eBid}_{\text{FastS}}$	72.09	15.02
$\text{JBoss}_{\mu\text{RB}}$ + $\text{eBid}_{\text{FastS}}$	72.42	16.08
JBoss + eBid_{SSM}	71.63	28.43
$\text{JBoss}_{\mu\text{RB}}$ + eBid_{SSM}	70.86	27.69

Table 7.4: Performance comparison.

The performance results shown here are within the range of measurements done at a major Internet auction service, where latencies average 33-300 msec, depending on operation, and average throughput is 41 requests/second per node [EBa04]. Since minimum human-perceptible delay is about 100 msec [Mil68], the increase in latency from using FastS to using SSM is of little consequence for an interactive Internet service like ours; latency-critical applications can use FastS instead of SSM.

7.5 Chapter Summary

In this chapter we described the results of experimentation with the J2EE prototype presented in chapter 6. We first asked whether microrebooting is effective in recovering from failures, and found it to be as effective as a process restart in the vast majority of cases, and in particular in all cases that represent dominant causes of downtime in practice. We then analyzed whether there is any benefit to using component-level microreboot instead of process restarts, and demonstrated that EJB-level microrebooting recovers 50x faster than JVM process restarts, with a 98% reduction in user-perceived downtime. In clusters, we found the benefits of microreboots to persist, in particular with respect to the preservation of session state and the load dynamics during failover. Finally, the performance degradation introduced by a crash-only design was found to be inconsequential when compared to a top online auction service.

In the following chapter we show that these results are more than mere quantitative improvements – they actually open the door for a qualitative change in recovery policies.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 8

Exploration of Microreboot-based Recovery Policies

The previous chapter showed microreboots to have significant quantitative benefits in terms of recovery time, functionality disruption, amount of lost work, and preservation of state and load dynamics in clusters. The low cost of microreboots engenders a new approach to high availability, that would be too expensive if full reboots were used.

In this chapter we evaluate, through quantitative experiments, the benefits offered by the liberal use of microrebooting in recovery policies for large-scale, failure-prone J2EE systems. We find that microreboot-centric policies afford considerable relaxation on the correctness constraints of failure detection (section 8.1); this further leads to a fortunate combination with the ability of microrebooting to be application-generic, thus enabling autonomous recovery (section 8.2). Experiments indicate that, in clusters, it is advantageous to attempt microreboot-based recovery prior to any node failover (section 8.3), which provides motivation for a more general approach of multi-tier recovery (section 8.4), in which microrebooting is always attempted first, prior to any other recovery mechanism. We also show how an entire system can be rejuvenated by parts, using what we call microrejuvenation (section 8.5). Finally, fast recovery offers the opportunity to mask microreboot-based recovery from end users by exploiting thresholds in humans' perception of response latency (section 8.6).

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

8.1 Lax Failure Detection

When recovery is cheap (i.e., fast and minimally disruptive), it becomes acceptable for a recovery manager to crash-recover suspect components even when it lacks the certainty that those components have indeed failed; the downtime risk of letting the components run may be higher than crash-rebooting healthy components.

In general, downtime for an incident is the sum of the time to detect the failure (T_{det}), the time to diagnose/locate the faulty component, and the time to recover. A failure monitor's quality is generally characterized by how quick it is (i.e., T_{det}), how many of its detections are mistaken (false positive rate FP_{det}), and how many real failures it misses (false negative rate FN_{det}). Monitors make tradeoffs between these parameters; e.g., a longer T_{det} generally yields lower FP_{det} and FN_{det}, since more sample points can be gathered and their analysis can be more thorough.

Cheap recovery could relax the task of failure detection in at least two ways. First, it allows for longer T_{det}, since the additional requests failing while detection is under way can be compensated for with the reduction in failed requests during recovery. Second, since false positives result in useless recovery leading to unnecessarily failing requests, cheaper recovery reduces the cost of a false positive, enabling systems to accommodate higher FP_{det}. Trading away some FP_{det} and T_{det} may result in a lower false negative rate, which could improve availability, since failures are not missed by the detection system.

8.1.1 Cheap recovery allows longer detection times

We illustrate T_{det} relaxation in figure 8.1. We injected a null reference fault in *BrowseCategories*, the main entry point for all browsing on the eBid Web site and, thus, the most-frequently called EJB (9.3% of total workload). The single-node system is loaded with 500 concurrent clients. We allow the system to recover with a process restart and compare the results to the same experiment using microreboot-based recovery. We delay recovery by T_{det} seconds, by introducing a delay in the monitoring system (shown along the horizontal axis); in this experiment, there are no false positives or false negatives. Note that a non-zero false negative rate is not interesting for any of the experiments in this section, because its effects are independent of the recovery technique – not triggering recovery is equally bad, regardless of what recovery was not triggered.

The two curves show how many requests fail as a function of the T_{det} delay. The horizontal dotted line indicates that, if a monitor took as long as 53.5 seconds to detect the failure, the system using microreboot-based recovery would still provide higher user-perceived availability than full

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

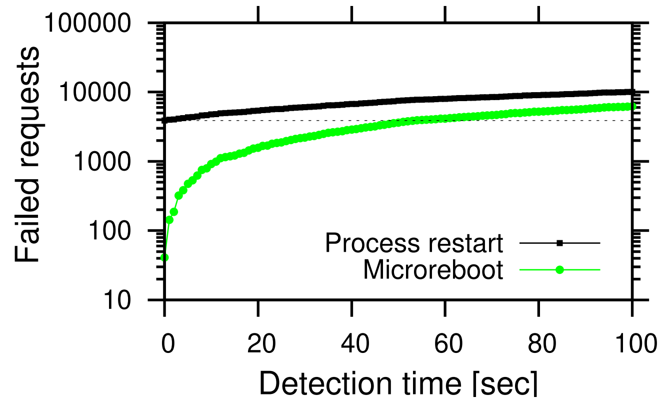

Figure 8.1: Relaxing failure detection: $FP_{\text{diag}} = 0$ constant, while T_{det} varies.

JVM restarts with immediate detection ($T_{\text{det}} = 0$). The two curves in the graph become asymptotically close for large values of T_{det}, because the number of requests that fail during detection (i.e., due to the delay in recovery) eventually dominate those that fail during recovery itself.

Doing real-time diagnosis, instead of recovering at the slightest hint of failure, has an opportunity cost. In this experiment, 102 requests failed during the first second of diagnosis. A microreboot averages 78 failed requests and takes 411-825 msec (Table 7.2), which suggests that microrebooting *during* diagnosis would actually result in approximately the same number of failures, but would additionally offer the possibility of curing the failure before diagnosis even completes.

8.1.2 Cheap recovery tolerates occasional mistakes

In figure 8.2 we explore the effect of false positives on end-user-perceived availability, given the averages from figure 7.1: 3,917 failed requests per JVM restart, 78 requests per microreboot. We keep $T_{\text{det}} \approx 0$, corresponding to UDP packet delivery time in our LAN; we vary FP_{diag} by generating spurious failure reports, tricking the recovery manager into believing there has been failure.

False positive detections occur in between *correct* positive detections; the false ones result in pointless recovery-induced unavailability, while the correct ones lead to useful recovery. The graph plots the number of failed requests $f(n)$ caused by a sequence of n useless recoveries (triggered by false positive detections) followed by one useful recovery (in response to the correct positive detection). A given number n of false positives in between successive correct detections corresponds to a $FP_{\text{det}} = n/(n+1)$, shown along the top horizontal axis.

The dotted line indicates that the availability achieved with JVM restarts and $FP_{\text{det}} = 0\%$ can

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

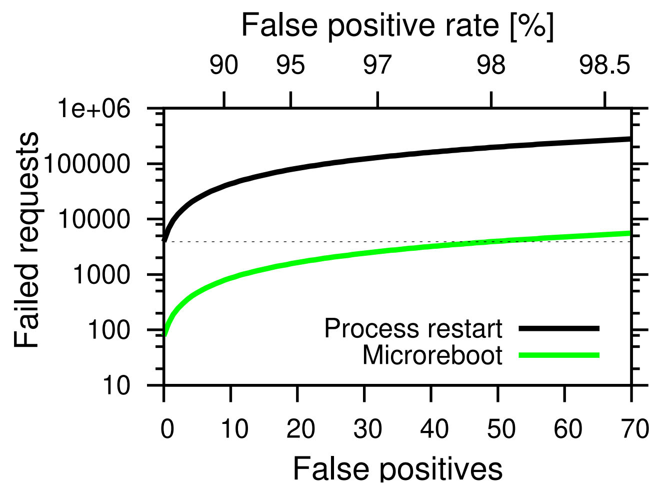

Figure 8.2: Relaxing failure detection: $T_{\mathsf{det}} \approx 0$ constant, while FP_{diag} varies.

be improved with microreboot-based recovery even when false positive rates are as high as 98%. This false positive rate is high compared to what can be achieved in practice; e.g., a fault decision tree-based diagnosis system used at eBay [CZL+04] achieves FP_{diag} = 24%. This suggests that a microreboot-based policy at eBay could improve availability, since their architecture and workload conditions are similar in nature to the ones we used in our experiments.

8.1.3 Summary

Engineering failure detection that is both fast and accurate is difficult. Microreboots give failure detectors more headroom in terms of detection speed and false positives, allowing them to reduce false negative rates instead, and thus reduce the number of real failures they miss. Lower false negative rates can lead to higher availability.

Microrebooting requires more precise diagnosis than process restarts, since microrebooting requires component-level precision. It seems, however, that the benefit of microreboots outweighs the added requirement in precision. For example, looking back to figure 7.1, our simple localization algorithm resulted in 6 microreboot actions instead of the necessary 3, thus exhibiting a FP_{diag} =50%; yet, microreboot-based recovery still reduced unavailability by a factor of 50. We would expect some of the extra headroom afforded by microreboots to also be used for improving the precision with which monitors pinpoint faulty components.

In general, we expect cheap recovery to blur the line between normal operation and recovery. When recovery becomes an order of magnitude cheaper, it allows one to think differently about

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

how and when to apply it. Since microreboots result in only a minor cost in goodput if applied by mistake, they provide the level of recovery performance needed to pursue a rationally-aggressive approach of initiating recovery at the slightest hint of failure.

8.2 Autonomous Recovery

Fast and safe microreboots allow us to apply more sophisticated failure detection policies, which is important because total recovery time is often dominated by fault-detection time [CAK+04]. One promising direction involves using statistical anomaly detection to infer failures. Such techniques have been shown to reduce time-to-detection at the cost of some false positives [KF05], and a simplified version of this approach has been successfully demonstrated in a state-management layer designed specifically to make reboots fast and safe [LKF04].

Automatic detection of failures coupled with fast recovery can improve availability over a scenario in which, while end users contact customer support (who then contact system administrators, who then re-establish the service's operation), the service stays down. Automatic detection could enable autonomous recovery, that would allow the system to recover on the scale of "machine time" rather than "human time," with the end result being a better service experience for its human end users and fewer support calls.

There are two important components in an autonomous recovery strategy: automatic detection and localization of faults, as well as a reliable form of recovery. Neither of these components can afford to be custom designed for the application, because they need to co-evolve with the system that is being cared for (through upgrades, workload changes, etc.). The increasingly large scale of today's Internet systems therefore calls for both detection/localization and recovery to be application-generic.

8.2.1 Application-generic failure detection

Microrebooting is an application-generic recovery technique, in that no a priori knowledge of the application is needed. Good failure detection, however, is generally application-specific. Writing application-specific monitors unfortunately does not scale, because of the difficulty of capturing all the possible wrong behaviors and changing the monitors in sync with changes to the monitored service. While developing detection and localization tools for rapidly evolving systems may seem difficult, our colleagues built an application-generic fault detection and localization program: Pinpoint [KF05] uses statistical learning techniques to detect and localize likely application-level

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

failures in component-based Internet services. Assuming that most of the system works correctly most of the time, Pinpoint learns a baseline model of system behavior; during system operation, it looks for anomalies (relative to this model) in low-level behaviors that are likely to reflect high-level application faults, and correlates these anomalies to their potential causes (software components) within the system. Application-generic detection in Pinpoint comes at the cost of occasional false positives.

We connected the Pinpoint prototype to our J2EE-based system to evaluate the possibility of autonomous recovery. Pinpoint instruments the JBoss EJB container to capture calls to the JNDI naming directory and the invocation and return data for each call to an EJB. It also instruments the Java Database Connection (JDBC) wrappers to capture information on interactions with the database tier. Each observation includes the monitored component information, IP address of the current machine, a timestamp, an ordered observation number, and a globally-unique request ID, used to correlate observations. All observations are sent to a centralized engine for logging and analysis. Anomalies in application behavior are detected by measuring the deviation between a single component's current behavior and the statistical model of normal behavior that was built over time; anomalies are found using the statistical χ^2 test of goodness-of-fit. For further details, we refer the reader to [CKKF04] and [KF05].

8.2.2 Recovering autonomously

In figure 8.3 we illustrate the functioning of the integrated system in reaction to a single-point fault injection; this is representative of the reaction to the other categories of faults we injected. We corrupted an internal data structure in *ViewItem*, setting it to null, which results in a *NullPointerException* for *ViewItem* callers. Labeled light-colored vertical lines indicate the point where the fault is injected and where the faulty component completes recovery, respectively.

To analyze the events that occur, we zoom in on the interval between 5:32 and 6 seconds in figure 8.4; the horizontal axis now represents seconds. We mark on the graph the points (along with the time, to millisecond granularity) at which the following events occur: we inject the fault (t_1), then the first end user request fails as a result (t_2), then Pinpoint sends its first failure report to the recovery manager (t_3), the recovery manager decides to send a recovery command to the microreboot hook (t_4), the microreboot is initiated (t_5), and finally the microreboot completes (t_6) and no more requests fail.

The system recovers on its own within 19.4 seconds of the first end user failure; of this interval, 18.5 seconds are spent by Pinpoint detecting and localizing the fault. The kind of faults we inject

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

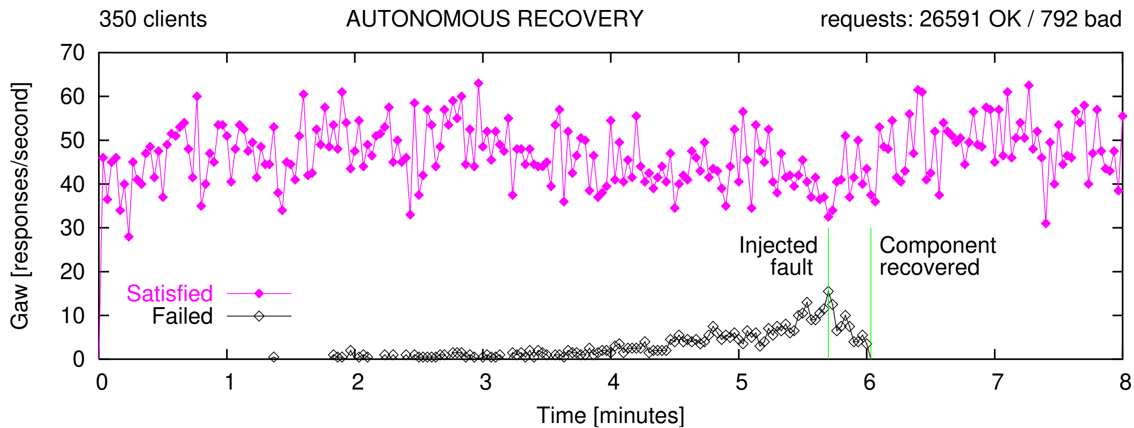

Figure 8.3: Autonomous recovery

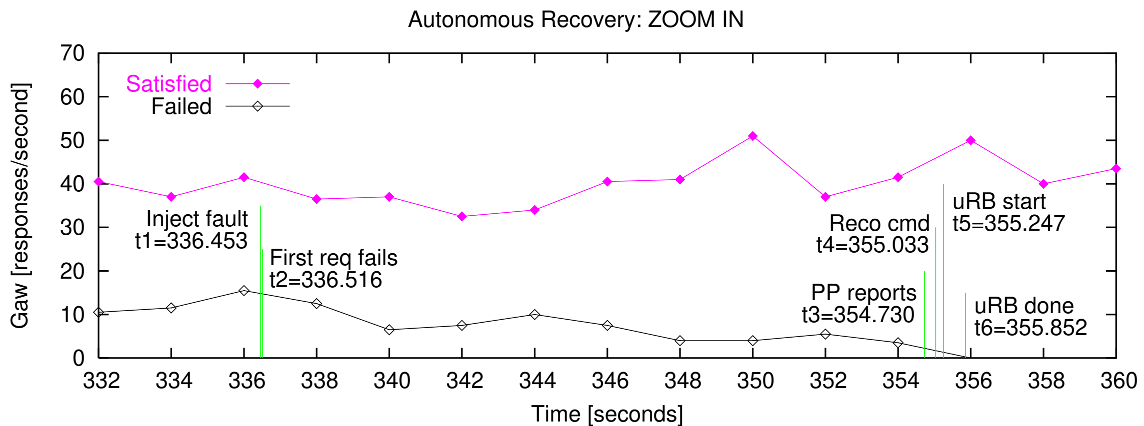

Figure 8.4: Autonomous recovery: zooming in on time interval [5:32, 6:00].

cannot be noticed by TCP or HTTP-level monitors, because the Web pages returned by the server constitute valid HTML. It takes on the order of a second or less to recover from a faulty EJB with a microreboot, compared to a full application reboot.

We also conducted a multi-point fault injection experiment: we simultaneously injected a data corruption *PutBid*, a Java Exception fault in *ViewUserInfo*, and a Java Error fault in *SearchItems-ByRegion*. Our system notices and recovers *PutBid* and *SearchItemsByRegion* within 19.9 seconds, while *ViewUserInfo* is recovered 46.6 seconds after the injection. The reason for the delay is that, in our workload, the former two components are called more frequently than the third. Thus, the Pinpoint analysis engine receives more observations sooner, which gives it a quicker opportunity to

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

detect and localize the injected fault.

8.2.3 Summary

These results argue for the combination of statistical anomaly detection with microrebooting to enable autonomous recovery. This combination is effective in detecting and recovering realistic transient faults, with no a priori application-specific knowledge. This approach combines the generality and deployability of low-level monitoring with the sophisticated failure-detection abilities usually exhibited only by application-specific high-level monitoring. Due to its level of generality, false positives do occur, but cheap recovery makes the cost of these false positives negligible. The synergy between Pinpoint and microreboots offers the potential for significant simplification of failure management.

Autonomous recovery enables automated response to failures, that operates in "machine time" rather than "human time," thus improving end user experience. Such autonomy is particularly useful for systems located in zero-administration environments, where access to the system is not immediate.

8.3 Alternative Failover Schemes

Since microrebooting is cheap, it makes sense to employ it instead of (or prior to) node failover in clusters. As seen earlier, node failover can be destabilizing: in the first set of experiments in section 7.3, failing requests over to good nodes while N_{bad} was recovering by microreboot resulted in 162 failed requests. In figure 7.1, however, the average number of failures when requests continued (*no* failover) being sent to the recovering node was 78. In that case, microrebooting without failover would have improved user-perceived availability over failover and microreboot.

The benefit of pre-failover microrebooting is due to the mismatch between node-level failover and component-level recovery. Coarse-grained failover prevents N_{bad} from serving a large fraction of the requests it could serve while recovering (as evidenced in figure 7.2). Redirecting those requests to other nodes will cause many requests to fail (if not using SSM for remote session state storage), or at best will unnecessarily overload the good nodes (if using SSM, the good nodes will have to spend extra time on contacting SSM, marshalling/unmarshalling objects, etc.). Should the pre-failover microreboot prove ineffective, the load balancer can do failover and have N_{bad} rebooted; the cost of microrebooting in a non-microreboot-curable case is negligible compared to the overall impact of recovery.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Assuming we microreboot first, we can use the average of 78 failed requests per microreboot instead of 162 and update the computation for six-nines availability from section 7.3.1. Thus, if using microreboots and *no failover*, a 24-node cluster could fail 683 times per year and still offer six nines of availability. Appealing to common engineering intuition, we conjecture that writing microrebootable software that is allowed to fail almost twice every day (683 times/year) is easier than writing software that is not allowed to fail more than once every 2 weeks (≈23 times/year for JVM restart recovery).

Another way to mitigate the coarseness of node-level failover is to use component-level failover; having reduced the cost of a reboot by making it finer-grained, *microfailover* seems a natural solution. Load balancers would have to be augmented with the ability to fail over only those requests that would touch the component(s) known to be recovering. There is no use in failing over any other requests. We expect microfailover accompanied by microreboot to reduce recovery-induced failures even further. Just like in the case of fault localization, microfailover requires the load balancer to have a thorough understanding of application dependencies, which could be inferred with approaches similar to AFPI (described in appendix A).

8.4 Multi-Tier Recovery

Microrebooting prior to failover in a cluster is a special case of a recursive recovery policy viewed as "multi-tier recovery": we try the cheap recovery first, because the opportunity cost of not attempting it can be high (in amortized terms). In this section we explore this idea more broadly.

The same way a software system cannot be flawless, neither can a recovery mechanism. Systems must therefore defend themselves in depth, with multiple layers of recovery, that can provide backup for each other. We believe the design space for recovery is spanned by two principal axes: cost of recovery and broadness of failures that can be recovered (figure 8.5). One axis captures the amount of coverage a technique can achieve in terms of failures it cures; the other axis refers to the general cost of employing that technique. In some sense, this set of axes captures a cost/benefit ratio for recovery, and multi-tier recovery policies constitute an exploration of this cost/benefit space.

Microrebooting is not a cure-all; component-level crash-restart works best on transient software and hardware failures and is effective against resource leaks and corruption of volatile data structures. Although these are important classes of bugs that are difficult to prevent with today's quality assurance processes, they do not represent all failures in systems. Other types (e.g., deterministic

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

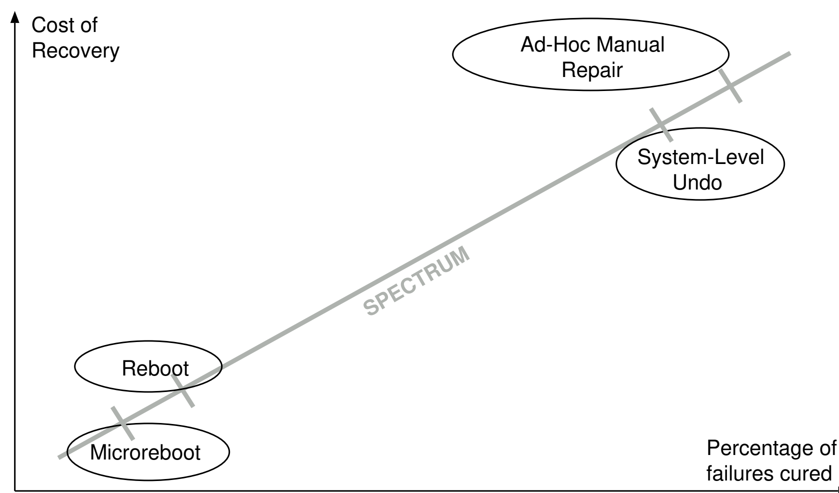

Figure 8.5: Abstract view of the design space for software recovery.

software bugs, corruption of persistent data) are seldom fixed by rebooting, while others (misconfigurations, botched upgrades) are not at all fixable by a reboot. In fact, some studies identify human-caused problems to be the largest single source of outages and data corruption in large-scale IT systems [OGP03]: incorrectly-performed upgrades, configuration changes that unintentionally disable or degrade service, inopportune component shut-downs, accidentally-deleted data, etc.

To extend recovery to all these types of failure, we can add a second line of defense based on the pattern of "undo/redo." Much like a user uses undo to fix typos in a word processor, the system operator can invoke *system-level* undo [BP03] to recover from state-corrupting failure and human operator error. Since undo is a much more comprehensive and broadly applicable recovery method covering the OS as well as user applications, administrators are willing to incur significantly higher overhead, since, in non-reboot-curable situations, the cost of not recovering (e.g., permanent data loss) is significantly higher than the cost of (possibly expensive) undo.

A system-level undo prototype [BP03] wrapped an Internet IMAP/SMTP email server with an undo/redo layer that uses a proxy to log all external interactions with the server (such as e-mail delivery and mailbox manipulation). Undo functionality is offered via a rewindable storage layer that can quickly restore a prior snapshot of all system state, including OS and application binaries as well as configuration state and user data. Redo is performed by replaying the logged external interactions via the proxy, thereby restoring end-user work lost during the undo operation while respecting the system-level repairs effected by the undo operation. For example, if an operator

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

misconfigured a global SPAM filter to drop legitimate mail, she could use undo to revert the configuration change then use redo to replay the lost email traffic, restoring the dropped messages to their rightful mailboxes.

A “defense in depth” approach to recovery consists of multiple layers that span the cost/benefit spectrum. A first line of defense should be a low-cost, low-overhead mechanism that has a good probability of repairing the problem, but a low opportunity cost in the event it doesn’t work. Microrebooting provides such a line of defense. We advocate a recovery policy in which microrebooting is always attempted first, at virtually no cost; if this does not cure the observed failure, more comprehensive (and potentially more expensive) recovery, such as undo, should be employed.

8.5 Microrejuvenation: Preventing Failures Caused by Resource Leaks

Despite automatic garbage collection, resource leaks are a major problem for many large-scale Java applications; a recent study of IBM customers’ J2EE e-business software revealed that production systems frequently crash because of memory leaks [MS03].

8.5.1 Resource leaks lead to failure

In figure 8.6 we illustrate the effect of memory leaks on our prototype application. We instructed *Item*’s container to leak 2 KB of memory on each call, and *ViewItem*’s container to leak 250 KB of memory per call; we chose these leak rates such that the experiment is completes in less than 30 minutes. The load was 350 concurrent clients. The amount of consumed memory increases until it exhausts all available memory, at which point the server is no longer able to answer requests, as it is spending most of the CPU cycles trying to reclaim un-reclaimable memory. The requests that fail are caused by attempted memory allocations that return an error. Once the server is completely hung, socket-level `accept()` calls hang in the Java networking layer. At t=25.2 and t=26, JBoss released an object pool, which freed up 4 KB of memory, allowing incoming requests to be accepted, but then promptly failed.

To avoid such leak-induced hangs or crashes, operators in industry resort to preventive rebooting, or software rejuvenation [HKKF95]. Some of the largest U.S. financial companies reboot their J2EE servers daily [Mit04] to recover memory, network sockets and file descriptors.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

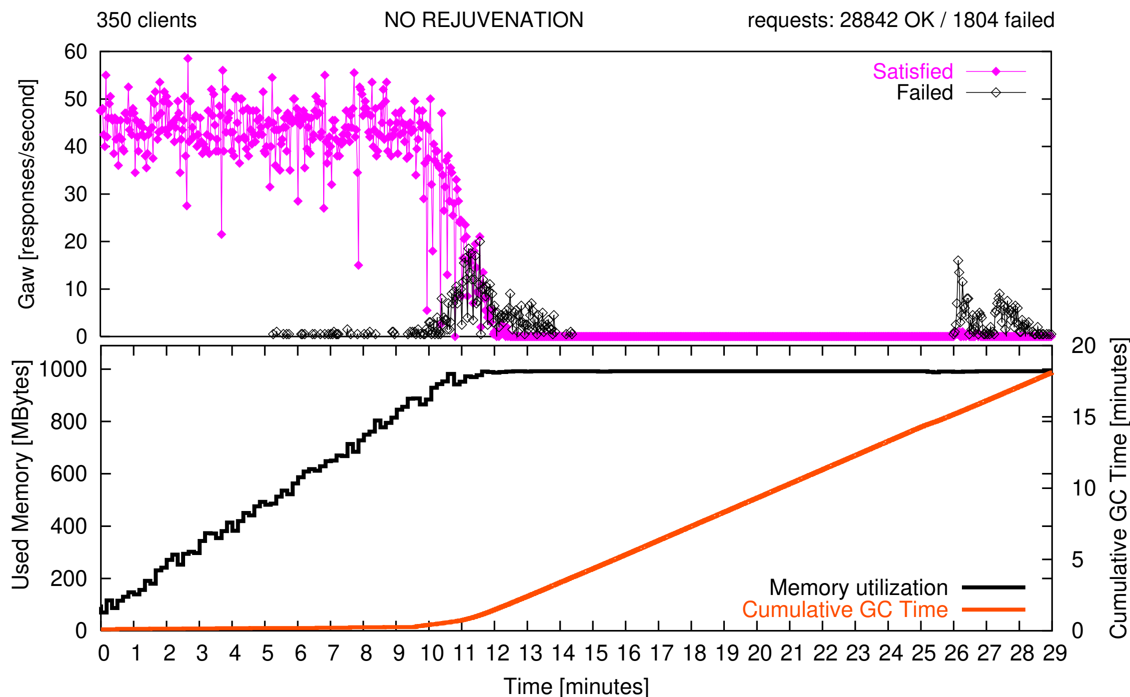

Figure 8.6: Memory leaks induce failure.

8.5.2 Rejuvenation prevents catastrophic failure

We wrote a server-side rejuvenation service that periodically checks the amount of memory available in the JVM; if it drops below M_{alarm} bytes, then it microreboots components in a rolling fashion (and invokes the system garbage collector) until available memory exceeds a threshold $M_{\text{sufficient}}$; if all components are microrebooted and $M_{\text{sufficient}}$ has not been reached, the whole JVM is restarted. This is a simple implementation; real production systems could monitor a number of additional system parameters, such as number of file descriptors, CPU utilization, lock graphs for identifying deadlocks, etc.

In figure 8.7 we illustrate the effect of rejuvenation using JVM process restarts, under the exact same conditions as the previous experiment: we injected a 2 KB/invocation leak in *Item* (an entity EJB part of the long-recovering *EntityGroup*) and a 250 KB/invocation leak in *ViewItem*. M_{alarm} was arbitrarily set to 35% of the 1-GByte heap (thus $\approx$ 350 MB) and $M_{\text{sufficient}}$ to 80% ($\approx$ 800 MB). Through rejuvenation, the necessary level of memory is automatically restored, well in advance of it causing the entire system to grind to a halt. In the experiment's 30-minute interval, a total of 4,317 requests failed, all due to the process restart; no requests failed due to insufficient memory.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

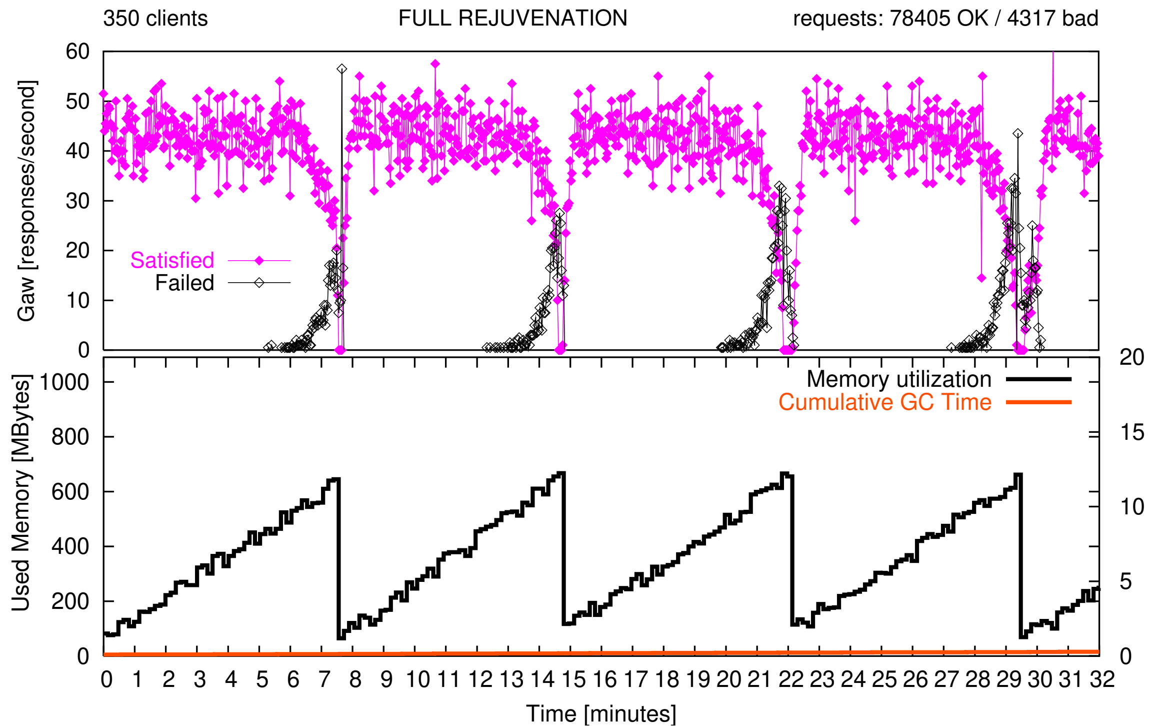

Figure 8.7: Full rejuvenation: T_{aw} and level of available memory.

8.5.3 Microrejuvenation

In this section we show that microreboot-based rejuvenation, or *microrejuvenation*, can be as effective as a JVM restart in preventing leak-induced failures, but cheaper.

We augmented the above-mentioned rejuvenation service with the ability to microreboot components when the M_{alarm} threshold is reached. Note that the service does not have any knowledge of *which* components need to be microrebooted in order to reclaim memory. We therefore devised a simple adaptive algorithm: the rejuvenation service builds a list of all components and, as components are microrebooted, it remembers how much memory was released by each one's microreboot. The list is kept sorted in descending order by released memory and, the next time memory runs low, the rejuvenation service microrejuvenates components starting at the front (thus microrebooting first those components that are expected to release most memory). The list is re-sorted using the updated released-memory amounts.

Figure 8.8 shows T_{aw} and the level of free memory during the microrejuvenation experiment. To be conservative, we examined the worst-case scenario for microrejuvenation, in which the initial

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

list of components has the two components leaking most memory at the very end; this way, the first time the rejuvenation service is triggered, it does not free up sufficient memory until it has microrebooted all components.

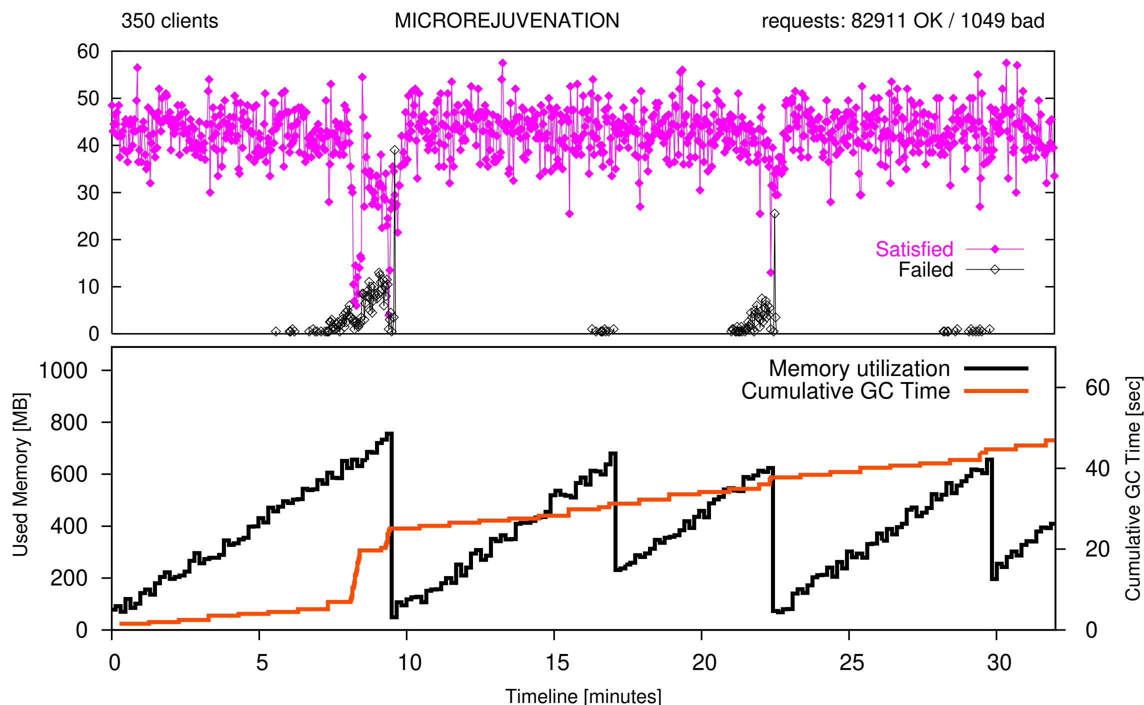

Figure 8.8: Microrejuvenation: T_{aw} and level of available memory.

During the first round of microrejuvenation (interval [7.43-7.91] on the timeline), all of eBid ends up rebooted by pieces. During this time, *ViewItem* is found to have the most leaked memory, and *Item* the second-most; the list of candidate components is reordered accordingly, improving the efficiency of subsequent rejuvenations. The second time M_{alarm} is reached, at $t = 13.8$, microrebooting *ViewItem* is sufficient to bring available memory above threshold. On the third rejuvenation, both *ViewItem* and *Item* require rejuvenation; on the fourth, a *ViewItem* microreboot is again sufficient; and so on. Overall, more time is spent in garbage collection, because we invoke the collector after every microreboot; in spite of this, the number of requests successfully served is higher than when full rejuvenation is performed.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

8.5.4 Summary

In the case of microrejuvenation, only 1,049 requests failed, representing a factor of 4x improvement over full rejuvenation. Moreover, T_{aw} never dropped to zero. The commonly used argument to motivate software rejuvenation is that it turns unplanned total downtime into planned total downtime; with microrejuvenation, we can further turn this planned total downtime into planned *partial* downtime.

Earlier, we described the application of Pinpoint [KF05] – the statistical anomaly detection engine – to the problem of failure detection. It appears to be a natural extension to use detected anomalies for the purpose of *anticipating* failures. Such anticipation can be used as a trigger for microrejuvenation.

8.6 User-Transparent Recovery

Fast recovery offers the opportunity to hide failure and recovery from end users. Various studies [BBK00, Mil68] have shown that, when a user is interacting with a computer system or service, response times up to 1-2 seconds are perceived as interactive; a delay of more than 8-10 seconds, however, leads the user to conclude the system has failed. In the case of Web services, exceeding the 8-second threshold leads the user to hit the Reload or Stop button, or click over to another site. These thresholds in the human perception of latency suggest that, as long as a Web site responds in less than 1 second, users will perceive it as interactive. If a Web site can recover from a transient failure and retry the failed in-flight request(s) within 8 seconds, affected users will have the illusion of continuous uptime – they will see a short delay rather than a failure.

Microrebooting an EJB takes less than 1 second, which permits us to use call retries to mask EJB failures from most end users. The HTTP/1.1 specification [GMF+99] offers return code 503 for indicating that a Web server is temporarily unable to handle a request (typically due to overload or maintenance). This code is accompanied by a `Retry-After` header containing the time after which the Web client can retry. This offers the opportunity to transform *MicrorebootInProgressException(n)* events into HTTP `Retry-After(n)` responses to Web browsers.

We implemented call retry in our prototype. Previously, the first step in microrebooting a component was the removal of its name binding from JNDI; instead, we bind the component's name to a sentinel during microreboot. If, while processing an idempotent request, a servlet encounters the sentinel on an EJB name lookup, the servlet container automatically replies with `[Retry-After 2 seconds]` to the client. We associated idempotency information with URL prefixes based on

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

our understanding of eBid, but this could also be inferred using static call analysis. We measured the effect of HTTP/1.1 retry on calls to four different components, and found that transparent retry masked roughly half of the failures, as shown in table 8.1 (data is averaged across 10 trials for each component shown). This corresponds to doubling perceived availability.

Operation / component name	**No retry**	**Retry**	**Delay & retry**
ViewItem	23	16	8
BrowseCategories	20	8	0
SearchItemsByCategory	31	15	0
Authenticate	20	9	1

Table 8.1: Masking microreboots with HTTP/1.1 `Retry-After`.

The failed requests visible to end users were requests that had already entered the system when the microreboot started. To further reduce failures, we experimented with introducing a 200-msec delay between the sentinel rebind and beginning of the microreboot; this allowed some of the requests that were being processed by the about-to-be-microrebooted component to complete. Of course, a component that has encountered a failure might not be able to process requests prior to recovery, unless only some instances of the EJB are faulty, while other instances are OK (a microreboot recycles all instances of that component). The last column in Table 8.1 shows a significant further reduction in failed requests. We did not analyze the tradeoff between number of saved requests and the 200-msec increase in recovery time.

A technique that achieves similar effects is that of using a *delay proxy*; we experimented with this approach in [CKK+03]. During the recovery process, we queue up incoming requests inside a proxy interposed between end users and the service in question; once recovery has completed, we replay requests that were in progress. This keeps clients from seeing failures due directly to the recovery process; instead of a failure, clients perceive an increased latency in the request. This induced latency is finite, as we stall requests for a maximum of 8 seconds, after which we return a failure to the client if recovery has not completed and the request cannot be admitted to the system.

The experiments reported here indicate a reduction by 50% in the user-perceived unavailability, when call-retry is in place. Such retries can be automated and performed transparently at multiple levels; we have only shown the highest level, as provided by the HTTP/1.1 `Retry-After` response header. At a lower level, calls between EJBs can be retried if a particular EJB is microrebooting, as described in chapter 5. We expect judicious and careful use of call-level retries to be a useful tool to improve availability in Web-connected applications.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

8.7 Chapter Summary

This chapter described ways in which microreboots enable more flexible and effective recovery policies; we provided a few examples of recovery and prevention actions that could be incorporated in a broader policy for recovering large-scale software systems. We showed through analytic argument and experimentation that a microreboot-centric policy affords relaxation on the correctness constraints of failure detection: microreboot-based recovery allowed the monitor in our prototype to take up to 53.5 seconds longer to detect a failure than the case where JVM restarts were used. In the same vein, moving from a recovery policy based on JVM restarts to one based on microreboots allowed the monitors to have a false positive rate in detection as high as 98%, while overall system availability was still higher. This level of tolerance for mistakes in detection allowed us to productively combine microrebooting with a detection system based on statistical analysis, which reduces the rate of false negatives at the expense of increased false positive rates; the result was autonomous recovery from microreboot-curable failures.

We argued for the use of a multi-tier recovery policy, in which microrebooting is tried first, at the slightest hint of failure, followed by other recovery techniques, if necessary. Should the microreboot cure the observed failure, it is a significant win in recovery time; should it be ineffective, the cost of having tried is insignificant. For illustration, we showed that, in clusters, attempting microreboot-based recovery can reduce the amount of unavailability perceived by end users

We introduced the notion of microrejuvenation, by which components are restarted prophylactically, before they actually fail. We showed that, compared to using process restarts, a microreboot-based rejuvenation policy in our prototype offers a 4x benefit in terms of unavailability.

Finally, we showed that fast recovery offers the opportunity to mask microreboot-based recovery from end users by retrying requests transparently and exploiting thresholds in humans' perception of response latency. In our prototype, we reduced unavailability resulting from online recovery by 50%, while maintaining user-perceived service quality.

In summary, we have shown that quantitative reduction of recovery time enables qualitative changes in the way such recovery is applied to failures in crash-only software systems.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 9

Limitations and Challenges

The success of microreboot-based recovery is predicated on several assumptions, and the absence of any of these can cause recovery to be less effective. In this chapter we present the limitations of microreboot-based approaches to recovery, along with challenges a system builder would face when applying the results of this work outside the domain of our prototype.

9.1 Results Proven in Java Environments

Our experimental evaluation was conducted on two Java-based systems: the JBoss/EBid online auction system and the Mercury satellite control system. We derived conclusions with respect to software systems in general by extrapolating the results shown in chapters 7 and 8. A more complete body of evidence supporting our thesis would consist of experiments run in more than one environment (e.g., also C/C++ programs using processes/threads for component boundaries, Microsoft .NET applications, etc.).

We expect our conclusions to hold in these programming environments as well; recent efforts in non-Java projects (e.g., a new version of Farsite [ABC$^+$02], new projects at Hitachi and IBM, the Nooks [SBL03] project) support this expectation. The crash-only design and the microreboot mechanism are not specific to Java, but rather use general software engineering techniques. We derived a programming model in which programs consist of independently-recoverable components that take a short time to restart and maintain important state in external, application-independent state stores. Structuring a C/C++ application, for example, as a collection of OS processes that can individually reinitialize very fast is a suitable way to achieve microrebootability in an otherwise-monolithic program; microrebooting such process-based components will be faster and less disruptive compared

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

to restarting the entire application or rebooting the operating system.

9.2 Microreboots Only Cure Transient Failures

Rebooting has little chance to be effective against persistent failures and certain classes of deterministic bugs; if a failure is not reboot-curable, microrebooting is unlikely to fix it. For a microreboot-based recovery approach to be successful, there need to be relatively more Heisenbugs than deterministic ones, so that restarting causes the failure to disappear once the program returns to its start state. We did find that, in present-day enterprise applications, the majority of failures that occur in practice are indeed transient in nature; there is not, however, sufficient evidence to argue that this assumption holds for most software. For example, failures in embedded control software may be more tightly related to non-transient physical events than to transient problems.

Even in crash-only systems, microreboot-based recovery is not a substitute for root cause diagnosis and bug fixing; performing such a substitution can be self-defeating. Without suitable failure analysis, feedback from the field back into development organizations is disrupted, reducing the rate at which software bugs are discovered and fixed. Whether employing microreboots or not, fixing a software bug will always improve the reliability and availability of a program (as long as no new bugs are introduced with the "fix").

We support a development approach in which components are explicitly designed to be microrebooted frequently and take conscious advantage of this expected mode of operation (e.g., by never explicitly releasing resources and instead relying on microrejuvenation to do so). We disapprove, however, of simply relaxing quality assurance processes and relying on microrebooting to cure whatever failures may occur during operation. Such relaxation would likely result in a relatively higher fraction of deterministic bugs, leading to a reduction in the effectiveness of microreboots.

9.3 Fine-grained Recovery Requires Fine-grained Workloads

In cases where a system's workload consists of long-running operations (such as long-running transactions), a microreboot can result in the loss of large amounts of work. It is sometimes possible to periodically microcheckpoint individual components [WKI00], to keep the cost of microreboots low, but this approach poses the risk of capturing reboot-curable state corruption in the checkpoint. Microrebooting clearly depends on workloads consisting of fine-grained, independent requests; if the units of work are small and little state needs to be preserved from one request to the next, then

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

there is little state that needs reconstruction upon the failure of one such request.

The benefits shown in our prototype are partly due to the general structure of Web-based applications, which is inherently favorable to microreboot-based recovery. HTTP shaped the evolution of all Web-connected applications, by forcing designers to piece together independent HTTP requests into sessions (through preservation of state at the server and the use of cookies as keys to that state). Applications that were normally accessed over persistent connections, as was the case with most enterprise applications at the time, migrated to an HTTP-based model. In the past decade, an increasing number of applications started using small transactions to turn previously-long-running operations into smaller, independently-recoverable units of work. With the advent of Web services, we expect this to become even more the case.

What developers of Web-based applications may have considered (and perhaps still consider) a headache, has ultimately turned out to be a tremendous benefit to high availability.

9.4 State Segregation Requires Discipline

It is usually not difficult to identify the critical state (i.e., that which cannot be recreated without requiring the end user to replay part of her interaction with the system). It is, however, difficult to ensure that such state is managed in a microreboot-safe manner, and we see this as the most important limitation to employing reboot-based recovery in existing systems. Code that was not designed to preserve the separation between important state and application code will often take "shortcuts" and perform unsafe updates that compromise the use of microreboots. In such systems, the engineering effort of untangling state management from application logic may present itself as too expensive.

In our experience with non-microrebootable J2EE applications (Petstore [Sun02], RUBiS [RUB], ECperf [Sub02]), we found the biggest challenges to be (a) separating code that manages session state from the rest of the application logic, and (b) ensuring that persistent state is updated using transactions. In general, however, J2EE applications are suitable for microrebooting and require minimal changes to take advantage of our microreboot-enabled application server, because the J2EE model encourages state externalization and component isolation.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

9.5 Achieving Good MTTF/MTTR Balance is not a Rigorous Process

We advocated in section 3.2.2 a good balance between MTTF and MTTR of components; frequently-failing components need to be recovered more frequently than seldom-failing ones, and thus should aim far fast recovery times. We described the application of this principle in the Mercury satellite ground station software, but did not specify how this could be done in the general case.

Unfortunately, the general problem of designing a componentized program that is well-balanced from the point of view of MTTF/MTTR cannot be cast (yet) as a rigorous optimization problem during the development phase. While for each subsystem there is certainly some "size" that provides the optimal trade-off between MTTF and MTTR, identifying this size must be done largely based on empirical data. A number of models have been developed over time (see [Lyu95] for examples) to help with MTTF prediction during the software development process, but few, if any, have proven successful in practice. Heuristics, however, can be useful – e.g., a mature piece of code can be assumed to have a higher MTTF than a new piece of code, so coupling a long-recovering component with the new code can be avoided.

Achieving the right MTTF/MTTR balance appears today to be more of an art than a science; we expect a practical approach to take into account not only directly measurable factors, but also development cost, maintenance costs, and the interplay of these measures with the resulting MTTF and MTTR.

If estimating the MTTF of a component ahead of time is difficult, then data from the field could be fed back into the development process to guide the redesign of programs. Runtimes designed to collect MTTF and MTTR data during operation and relay it back to the vendor can be very useful in this respect; many software vendors are building such features into their runtimes (e.g., Microsoft Windows XP). Unfortunately, small code changes in the component itself or in peer components can lead to drastic changes in MTTF; as a result, the MTTF can change significantly over time, eliminating the balance achieved in a prior release of the software.

9.6 Componentizing Legacy Software is Difficult

In order to perform fine-grained microrebooting, applications need to be composed of fine-grained components that can be restarted independently of each other, while important state needs to be managed by specialized state stores. However, most software that is currently in operation is monolithic, and microrebooting cannot be applied safely to such non-crash-only applications. Some

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

legacy programs are fairly easy to transform into microrebootable ones, while others are not.

In the case of all software, whether monolithic or not, there exist boundaries along which subsystems can be isolated from each other – these boundaries are generally suggested by the functionality of the program in question. Enforcing these boundaries can be done with low-level mechanisms, as would be the case with separate process address spaces or the use of virtual machines. Some research projects have explored the possibility of wrapping harnesses around parts of legacy software (e.g., [KPGV03]).

The challenge resides in making these components sufficiently small to obtain a noticeable benefit from microrebooting. In our experience, mature monolithic programs require non-negligible amounts of redesign prior to being broken down into fine components; strongly modularized programs present more attractive opportunities for componentization.

We are encouraged however by the fact that componentized software design has been gaining in popularity for many reasons beyond recovery. Componentized programs offer flexibility in development, packaging, and testing; they enable fine-grained upgrades while in operation, thus reducing the cost of these upgrades; and components can be distributed, replicated, and/or scaled individually. We have already witnessed several major rewrite projects, in which commercial monolithic software underwent componentization for one or more of these reasons; such projects have the added benefit of making microreboots possible.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Chapter 10

Conclusions

This dissertation defines the crash-only design, a way to build software systems that can be recovered safely and fast using fine-grained restarts. Crash-only design is centered around modularity and the complete separation of process recovery from data recovery, with the delegation of the latter to specialized state stores. This design differs from prior approaches primarily by viewing rebooting/restarting of programs as a necessarily-routine operation in highly available software. Although crash-only design may introduce performance overheads, we argued that the benefits outweigh the drawbacks; in our J2EE prototype, we showed the overheads to be negligible compared to an equivalent deployed system.

We define the microreboot mechanism, a way to make system recovery cheap by reducing the scope of a system reboot to those components that are indeed faulty. We build upon prior work by casting most failures into reboot-curable problems and effectively recovering from them. For example, in our J2EE prototype, microrebooting cures the majority of failures empirically observed to cause downtime in real-world Internet services. Compared to recovery based on JVM process restart, microrebooting is an order of magnitude faster and an order of magnitude less disruptive, even in multi-node clusters.

The use of microreboot-based recovery simplifies failure management policy, thus improving its probability of success; once recovery becomes fast and safe, it has even further reaching effects. Using microreboots, we reclaim memory leaks in our prototype application without shutting it down. Microreboot-based recovery achieves higher levels of availability even when fault detection has false positive rates as high as 98%. Microrebooting faulty nodes in clusters of J2EE application servers improves availability even over the traditional "failover and reboot" approach. Finally, 50% of microreboots are masked using simple, transparent call-level retry, unbeknownst to the system's

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

end users.

In summary, crash-only design in conjunction with microreboot-based recovery mitigates several key dependability problems, such as the difficulty of building reliable large-scale software, the complexity of managing failures in large infrastructures, and the effects of long diagnosis times on availability. We conclude that our approach is an appropriate way to structure software with high-availability requirements, and we advocate the use of microrebooting as a first-line defense against transient failures.

The premise of this dissertation is that many forces prevent large-scale software from being reliable; accepting software failure as a fact, we showed that structuring systems for fast, minimally-disruptive recovery is practical and makes systems highly available. The fact that we focused on fast recovery instead of trying to eliminate failures altogether allowed us to increase the availability in a prototype system by several orders of magnitude.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Appendix A

Automatic Failure-Path Inference

In complex but modular software systems, it is useful to have a failure dependency graph, to understand how a failure in one part of the system will propagate to or affect other parts. Such dependency information has been used for root cause analysis, for guiding automated system recovery, and for identifying where debugging efforts should be focused.

Automatic Failure-Path Inference (AFPI) is an application-generic, automatic technique for dynamically discovering the failure dependency graphs of componentized Internet applications. AFPI's first phase is invasive, and relies on controlled fault injection to determine failure propagation; this phase requires no a priori knowledge of the application and takes on the order of hours to run. Once the system is deployed in production, the second, non-invasive phase of AFPI passively monitors the system, and updates the dependency graph as new failures are observed. This process is a good match for the perpetually-evolving software found in Internet systems; since no performance overhead is introduced, AFPI is feasible for production systems.

We applied AFPI to J2EE and tested it by injecting Java exceptions into an e-commerce application and an online auction service. As will be shown later, the resulting graphs of exception propagation are more detailed and accurate than what could be derived by analysis of readily-available static application descriptors.

The rest of the appendix is organized as follows: Section A.1 describes the specifics of our approach, section A.2 presents experimental results validating the accuracy of our f-maps and confirming that there is no performance overhead in applying AFPI. We highlight specific cases in which AFPI finds interesting fault propagation paths that would have been difficult to find manually, as well as cases in which it eliminates paths that appear in static dependency graphs but do not actually propagate faults in practice. Finally, section A.3 concludes.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

A.1 Approach

AFPI consists of two phases. In the (invasive) staging phase, the system actively performs fault injection, observes its own reaction to the faults, and builds the f-map. In the (non-invasive) production phase, the system passively observes fault propagation when faults occur during normal operation, and uses this information to evolve the f-map on an ongoing basis. In this appendix we focus on the staging phase.

Aside from the reasons described in the main body of the dissertation, our choice for a J2EE application server in the case of AFPI was also motivated by the fact that Java offers features such as reflection and other mechanisms that allow AFPI to discover relevant faults to inject. J2EE enforces a particular application structure in which components have a small number of well-defined entry points, and by instrumenting the platform we can intercept communication between components and, hence, propagation of faults.

A.1.1 AFPI algorithm

The general algorithm for the staging phase is as follows:

1. Initialize a global fault list to be empty.

2. Start the application and any external components it uses (e.g., a separate database).

3. Every time a new component C of the application is deployed (whether at startup or during the operation of the application server), use reflection to discover the methods exported by its interface.

4. For each method M_i of C, use reflection to discover the set **F** of Java exceptions declared as throwable by M_i, and for each $F_j \in \mathbf{F}$, add the triplet (C, M_i, F_j) to the global fault list.

5. Also add to the global fault list any exceptions corresponding to "environmental" faults that could occur during execution of method M_i. The set **E** of all such environmental exceptions could include network-related exceptions, disk I/O exceptions, memory-related exceptions, etc. In our current approach, we add a triplet (C, M_i, E_j) for *every* exception type $E_j \in \mathbf{E}$. Injecting application-defined exceptions stresses robustness to application bugs, while injecting environment-related exceptions primarily probes the paths through which application-external faults can propagate.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

6. Once all components have been deployed, select a triplet (C, M, F) from the list of faults, and arrange to inject exception F into component C the next time method M is called. Section A.1.2 describes how this is done.

7. Start the load generator (client emulator), to exercise the application.

8. As failures occur, the monitoring agent is notified by a separate fault detector. As the monitor receives notifications, it builds up an *f-map* for each type of fault. An f-map is a directed graph that captures failure propagation: the presence of a directed edge (u, v) means that a fault in component u propagated and caused component v to fail. If, however, u's fault propagates but is properly handled by v, then no externally visible behavior is reported to our monitor, so edge (u, v) is not added to the f-map. F-maps for different fault types may differ, because some faults propagate from the callee up the call tree and others do not. In all our experiments, the injected faults always resulted in observed failures, i.e., exceptions that propagated to at least one component.

9. Save the current f-map and list of faults to stable storage, shut down and restart the application, and continue with the next (C, M, F) triplet. The fault injection experiments end when the list of faults has been exhausted. We restart the application between injections to avoid spuriously representing cascading failures in the f-map.

Two notable differences emerge in comparing our fault injection approach to other recent work. First, in step 4, we directly induce application-visible exceptions, in contrast to prior work that injects low-level hardware faults. Determining which (if any) application-visible failures result from low-level hardware faults requires the construction of a fault dictionary [KIR+99], which has proven difficult.

Second, some recent work [Fea03] attempts to narrow the possible faults injected at a particular point based on static or dynamic analysis; for example, if a particular bytecode sequence is known to specify a read from a network stream rather than from a file, one might inject only network-related hardware errors rather than low-level disk errors at that point. As stated in step 5, we avoid such narrowing and simply enumerate all possible environmental conditions that can be expressed as Java exceptions. Besides the fact that deriving more specific information using static analysis can be cumbersome, we would like to minimize our assumptions about whether, in fact, the executing method would really be unaffected if an “unrelated” low-level fault occurred (consider a method that does no disk I/O, but a low-level disk fault occurs while that method is trying to page in data

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

or code). In fact, all we assume is that it is *possible* that a given exception might be thrown by a particular method. We return shortly to the question whether most low-level faults do in fact manifest as application-level exceptions.

A major flaw in many fault injection experiments in the literature is that they do not account for correlated faults. However, in large scale production systems, true independent failures are rare [Bar02, Ach02]. Therefore, once the AFPI sequence of single-point injections completes, our system does multi-point injections, to simulate correlated failures. The monitoring agent adds to the f-map any additional edges that it detects this way.

Once the multi-point injection phase completes, the system can be deployed into production. The monitoring agent continues to observe the system's reactions to "naturally occurring" (i.e., non-injected) faults, and modifies the f-map based on these observations. Whenever an edge in the graph is added or re-observed, a timeout is reset on it; propagation paths that do not manifest for a long time are removed from the graph. Thus, the f-map is a continuously evolving representation of the application. This passive phase is in no way dependent on the active, fault-injection phase – it would work even without an initial draft of the f-map, but it would take much longer to converge onto a correct representation of the dependencies. We can therefore think of the fault injection phase as an optimization.

A.1.2 Modifying JBoss to enable AFPI

JBoss consists of a microkernel, with the various services being held together through Java Management Extensions (JMX). The services are hot-deployable, which implies that one can replace an existing service with our modified version at runtime, and the server will properly reintegrate it. We built our failure monitoring and fault injection facilities as two separate services of the JBoss microkernel, so that failures are detected independently of any specific knowledge that they are being injected.

In addition, we modified existing JBoss code in three ways. First, whenever a new EJB is deployed (i.e., the EJB is instantiated in a new container), the fault injector uses Java reflection to enumerate the EJB's methods and the exceptions each method can throw, in addition to those thrown by the JVM itself (i.e., step 4 of the AFPI algorithm). This process is identical whether the EJB is deployed as part of regular application deployment, or as part of a live upgrade or bug fix.

Second, we provide a new container method by which we can instruct the EJB that the next call to method M should throw exception X, i.e. step 6 in the algorithm. During the fault injection stage, the fault injector will systematically inject every kind of exception that can be thrown by

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

each method in the EJB, one at a time. It will attempt to throw both declared exceptions (i.e., those explicitly declared by the EJB's Java method signature, some of which may be handled using `catch` and others which may be passed up to the caller) and non-declared exceptions (e.g., a runtime condition such as *OutOfMemoryError*, to simulate more generic system-level problems that most EJBs would not try to detect directly, as in step 5 of the algorithm).

Third, we modified the EJB container so that, when an exception is thrown by an EJB, the stack trace is parsed and we extract the identity of the calling component (EJB/servlet/JSP) and the invoked component. This information is passed to AFPI, which uses the information to build up a failure propagation map.

Since we modified the application server rather than a specific application being deployed, we can invoke the new functionality on *any* J2EE application that runs on JBoss.

A.2 Experiments

Our experiments address three issues: the suitability of our injected faults, the accuracy and usefulness of our f-maps, and the cost in terms of time and performance at which we can obtain these f-maps.

All experiments, except performance overhead, were performed on an Intel Pentium–4 1GHz machine, with 512 MB RAM, running Linux 2.4.9. We modified JBoss 3.0.3 running on Sun Java 2SE 1.4 and Sun Java 2EE 1.3.1. The applications we chose for our experiments are Petstore 1.1.2 and RUBiS 1.3. Petstore is a freely available sample J2EE application from Sun that implements an e-commerce site where users can maintain an account, update their profile and payment information, browse a merchandise catalog, add/remove items to/from a shopping cart, complete a purchase, etc. It consists of 233 Java files and about 11K lines of code. RUBiS, developed at Rice University, implements a Web-based auction service modeled on eBay. It contains 582 Java files and about 26K lines of code. For each of these applications, the invasive staging phase took between two and three hours to complete.

A.2.1 Suitability of injecting Java exceptions

Since we intend for our technique to be used on real systems, it is important that we inject faults representative of those that would occur in real systems. We inject application and JVM-visible faults (exceptions, application-visible or OS-visible resource exhaustion) rather than low-level faults (bit flips, stuck-ats) for three reasons. First, low-level faults are not that common in Internet systems,

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

unlike, e.g., space-born applications exposed to radiation; Internet services go down mainly due to software bugs and operator errors [MG95, Cho97, Sco99, AIS+01, OGP03]. Second, faults not corrected by the hardware will often manifest as some higher-level fault, but if transient, the mapping may appear nondeterministic. A "random" bit flip *may* corrupt an important/live data structure or trash the heap, but in many cases turns out to be harmless [CMB+01]. Getting reliable failure propagation for this type of faults is difficult; we are not aware of prior work that systematically addresses this problem.

An important question concerns coverage of failures: have we exercised all the conditions that could trigger a particular exception, especially given that the way the exception is handled may depend on the particular state of the program at the time the exception occurs? Although the experiments performed so far have not provided evidence for this, we anticipate that our coverage is not complete. However, in cases where partial recovery fails due to incorrectly determining the subset of components that must be recovered, a full reboot could always be used.

Determinism is related to the timing of our injections: we inject an exception after the applications has gone through a sequence $x_0, x_1, ..., x_k$ of state transitions; if we were to inject, instead, after a sequence $x_0, ..., x_k, ..., x_n$ the application could conceivably handle the exception differently. We have found it to not be the case in the two applications we looked at; we also verified it experimentally, by injecting the same set of exceptions in different orders and with different timings, but the resulting f-maps were the same. However, in the more general case, it is quite possible that a Java application would treat an exception differently. Such differentiated treatment of failures would lead over time to denser f-maps, because all the alternate paths would be captured simultaneously.

In addition to injecting the kinds of faults the application designer originally thought of, we choose faults that are in fact commonly observed as software-related transients. Exceptions explicitly declared by the EJBs correspond to the designer's own knowledge of particular "expected" failure modes. With respect to undeclared exceptions, it is reasonable to ask whether the kinds of failures that are independent of application semantics – OS-level resource exhaustion, network connectivity problems, manifestation of bugs in the OS kernel or libraries – are indeed manifest as Java-visible exceptions.

When an internal error or resource limitation prevents the Java virtual machine from implementing the semantics of the Java programming language, it throws an exception that is a subclass of VirtualMachineError; these exceptions are included in table A.1.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Exception	**Possible real life cause**
declared exceptions	application-expected faults
OutOfMemoryError	memory exhaustion
StackOverflowError	code bugs
IOException	failed or interrupted I/O operations
RemoteException	remote method invocation failure
SQLException	database access error
NullPointer	code bugs and data errors

Table A.1: Exception types used in AFPI experiments.

We also injected timing faults, in which calls to various EJBs were delayed, but due to the blocking nature of RMI (Java's RPC-like remote method invocation mechanism), such timing faults did not induce any failure in Petstore or RUBiS.

We did not find literature that documents the extent to which different JVMs actually translate such low-level faults into Java-visible exceptions. We performed a number of ad hoc experiments (see table A.2 for a few examples) to determine whether Java exceptions were indeed a reasonable way to simulate such faults. The results were satisfactory: using the Sun HotSpot JVM, all faults we injected at the network level (e.g., severing the TCP connection), disk level (e.g., deleting the file), memory (e.g., limiting the JVM's heap size), and database (e.g., shutting DBMS down) resulted in one of the exceptions shown in table A.1. Our validation, however, is certainly not exhaustive.

Induced failure	**Exception**
bad server in registry	ConnectException
RMI registry unreachable	ConnectException
server not in registry	NotBoundException
server crashes during call	UnmarshalException

Table A.2: RMI experiments evaluating the conversion of real faults into Java exceptions.

A.2.2 F-maps compared to existing structures

In this section we examine whether our f-maps are as accurate as those obtained using other techniques, and, if so, whether they are any better. We compare f-maps obtained through our introspective method with a dependency graph built by manual inspection of J2EE deployment descriptors – a programmer-supplied XML document for each J2EE component, that describes a component's deployment settings. An EJB's descriptor declares, among other things, what other EJBs this EJB calls; this suggests using the collection of descriptors as an approximation of the static call graph.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

In our implementation, the nodes in an f-map represent EJBs, servlets, JSPs, and special data access objects (DAOs) used for accessing databases. Figure A.1 shows two versions of the Petstore f-map: one derived by manually inspecting Petstore's deployment descriptors (top f-map) and one obtained automatically with our introspective system (bottom f-map). The bold edges are those that are not common between the two f-maps.

Notice two types of differences between the f-maps: First, the injection-based f-map is missing some edges present in the descriptor-based f-map: AccountEJB → OrderEJB, CatalogEJB → ShoppingClientControllerEJB, and EStoreDB → Web tier. Second, the injection-based f-map has additional nodes and edges that are not present in the descriptor-based f-map: HttpJspBase, MainServlet, and six JSPs, with the corresponding edges. The deployment descriptors group all Web components (servlets and JSPs) into one entity, called the Web tier.

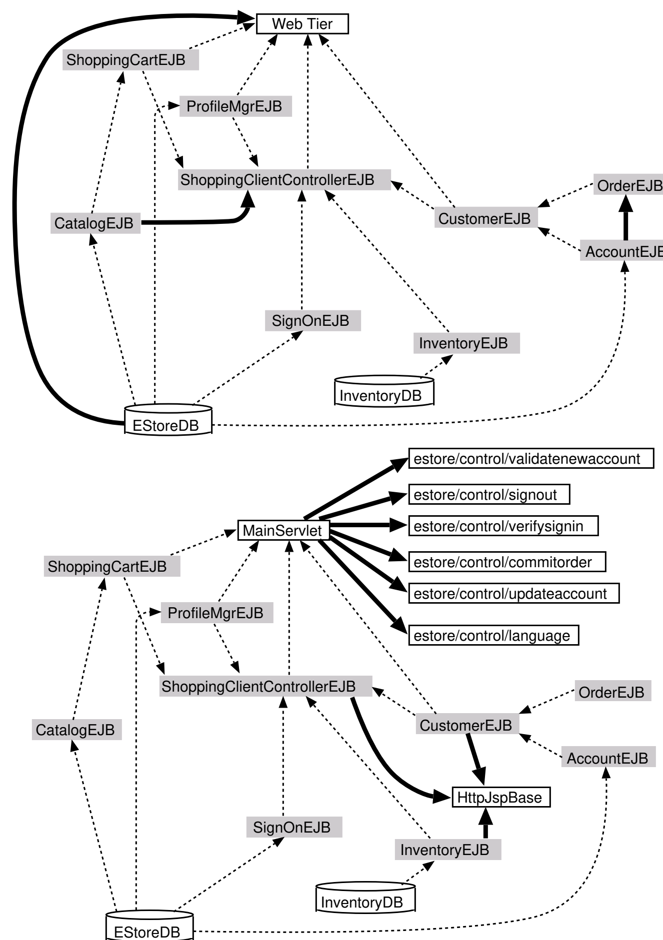

Figure A.1: F-map for Petstore: deployment descriptors (top) vs. AFPI (bottom).

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

We were most concerned about the missing edges, because they seemed to imply that our technique failed to discover existing dependencies. However, upon inspection of the code, we found that the injection-based f-map was correct. In the case of the AccountEJB $\rightarrow$ OrderEJB edge, OrderEJB did indeed maintain a reference to AccountEJB, but it never interacted with that EJB, hence no opportunity for a fault propagation from AccountEJB to OrderEJB. In the case of CatalogEJB $\rightarrow$ ShoppingClientControllerEJB, ShoppingClientControllerEJB did not even have a reference to CatalogEJB, so the deployment descriptors were simply wrong. This illustrates that, contrary to our initial expectations, the descriptors cannot be relied on for understanding the structure of the application.

Finally, the EStoreDB $\rightarrow$ Web tier edge is present in the descriptors due to a servlet which runs at setup time and populates the database with default information (users, merchandise, etc.). This servlet is run once at installation and never again during normal operation, which is why our f-map correctly indicates there is no direct fault propagation edge from the database to any of the Web components.

The second difference is that components are discovered at a finer grain than can be captured in the deployment descriptors. For example, what the descriptors showed as the "Web tier," our system dissected into the individual servlets and JSPs, along with fault propagation information. These additional f-map nodes naturally result in new edges along which faults can propagate, edges that can be used in better pinpointing the source of failure and thus provide a more powerful tool in deciding which components to recover. An interesting case is that of the HttpJspBase node; we tried to find it in the Petstore source code to understand its role, but it turned out it is not part of the application. HttpJspBase is the superclass of all JSP-generated servlets and it is part of Jetty, the servlet/JSP container that we used with JBoss. Hence, our technique was able to identify interactions with components which, although not part of the application, are still parts of the system delivering the service.

Obtaining the f-maps for RUBiS, as indicated in figure A.2, confirmed the properties observed with the Petstore f-maps. This time, the deployment descriptors turned out to be quite conservative, in that the AFPI-based f-map contained strictly more information than the descriptors. Many of the inter-EJB dependencies fail to be indicated as references in the deployment descriptors. Dependencies between EJBs and servlets are not captured in deployment descriptors either. While much of this information could be obtained through static analysis, certain conditions cannot be found, such as code that is dead, owing to a dependency on the current date.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Figure A.2: F-map for RUBiS: deployment descriptors (top) vs. AFPI (bottom).

AFPI needed no application-specific knowledge to derive the correct set of dependencies between components, and didn't even require access to the application's source code. As was mentioned previously, the collection of deployment descriptors does not need to reflect a correct static call graph in order for the application to run, and in fact one would expect EJB evolution to make erroneous descriptors more common. This is especially true given that developers often maintain the descriptors manually [Are02], although commercial tools are available to programatically generate deployment descriptors from EJB sources. Moreover, even if the deployment descriptors were correct and complete, they would capture all possible paths that *might* propagate faults at runtime, whereas we are interested in those that actually *do* propagate faults.

Since deployment descriptors fail to provide an accurate static call graph, we considered obtaining the call graph using a tool that directly inspected the source code. Of course, such a call graph would have to be significantly pruned in order to make it useful; the static call graph is normally exponential in the program size, and researchers currently attempting to apply static analysis to improve fault coverage admit they will have to use a smaller-sized approximation of the full static call graph [Fea03]. Even if we could work with a compact representation of the static call graph, it would show paths along which faults might propagate, not only the ones along which faults do

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

propagate, and it would not reveal edges that propagate faults unrelated to individual method invocations, such as an errant thread that overallocates memory and causes another unrelated thread in the same JVM to get an out-of-memory exception.

A.2.3 Fault-class-specific f-maps

Unlike general dependency graphs, our technique allows the system to obtain and maintain separate f-maps for each fault class. Such maps enable targeted recovery in the case of specific faults. Since one motivation of obtaining f-maps is fast recovery, we do want such fine-grained information about failure propagation so that we can recover the minimal subset of failed components. By eliminating edges that do not reflect actual (observed) propagation paths, we can do more efficient microrecovery.

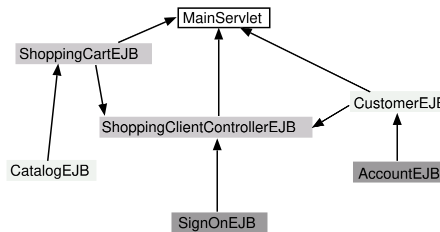

Figure A.3: Petstore f-map based exclusively on application-declared exceptions.

Figure A.3 shows a Petstore f-map discovered by injecting exclusively application-declared exceptions. Such an f-map may help us in deciding how useful it would be to suppress environmental/external sources of faults (e.g., by purchasing more reliable hardware). Notice that this f-map is considerably simpler than the previous f-maps.

We also examined Petstore's five fault-class-specific f-maps for undeclared exceptions, i.e., those obtained by injecting *OutOfMemoryError*, *StackOverflowError*, *IOException*, *RemoteException*, and *SQLException*, respectively. To our surprise, each of these f-maps was virtually identical to the cumulative one in figure A.1, which represents the union of all the fault-class-specific f-maps. Inspecting the Petstore code provided a simple explanation: the application contains almost no code to handle exceptions that may occur from its interaction with the environment. Such lack of robustness causes any external exception to appear as "something bad" that the affected EJB does not handle gracefully.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

A.2.4 Correlated faults

As described earlier, the second phase of f-map generation consists of injecting correlated faults. The algorithm for this second phase is similar to the first phase, with one exception: in order to generate the fault list, we take the Cartesian product of the original fault list with itself, and then eliminate any 6-tuples for which the same component is involved in both the first and the second fault of the tuple. This gives all possible combinations of 2 faults that involve distinct components. An analogous algorithm is used for building the list for n-point injections, where $n > 2$. If new propagation paths are discovered through multi-point injection, they are merged into the f-map.

We were particularly interested in verifying the robustness of the information in the f-maps to correlated fault scenarios. The f-maps converged much quicker onto their final form, because of the richer set of faults that the system was exposed to. The unexpected result, however, was that we obtained f-maps identical to those resulting from single-point injection. While this might suggest the f-maps are robust, we actually believe the result is due to another reason: in a request/response system with very little recovery code, when a request enters the system and hits a faulty component, it will almost always fail at that point and not proceed further through the system. Hence, the correlated faults are likely being consumed by separate requests, as if the faults had been injected separately.

A.2.5 Performance overhead

We evaluated the performance impact of our modifications by comparing the performance of unmodified JBoss to that of our instrumented version, with the same Petstore workload we used for generating the f-maps. We found no statistically significant difference in performance when no exceptions are thrown, i.e., under steady state operating conditions. This suggests that it is feasible to implement our technique in a production system.

JBoss and the database were hosted on the same Intel-P3 450MHz-based machine with 512MB of RAM. The workload generators ran on a dual P3/1GHz machine with 1.5GB of RAM. The purpose we chose machines with such different characteristics was to allow us to saturate the server. Both machines ran Linux 2.4 with Sun's JVM 1.4.1 and were connected via 100Mbps Ethernet. The server's CPU was saturated at 100% during each run. Our modified JBoss completed each test run on average in 93 seconds, compared to 94.8 seconds for unmodified JBoss; standard deviation was 5.8 in both cases. We consider this improvement in the running time of the application to be just statistical noise, as we do not think any of our changes would make JBoss run faster.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

A.2.6 Weaknesses

A problem specific to our experiments is the fact that we only injected exceptions and timing faults. First, there may be low-level faults that do not manifest as exceptions in the Java VM, although we haven't found any in our ad hoc experiments. Second, we haven't explored yet data-level faults, in which a called component provides wrong data. Most of these faults require application-specific knowledge to detect, unless we transparently employ redundancy combined with a voting scheme. A special kind of data faults are null pointers, which we inject using NullPointerException.

Problems also arise from the fact that we currently do not collect all the information we could about the exceptions, thus preventing us from distinguishing between propagation paths composed of a sub-path that propagates fault X, chained to a sub-path that propagates fault Y. We do collect per-fault-class f-maps, but in the case of application-declared exceptions, our current version of the system would indicate the entire path as a fault propagation path, without discriminating among the faults.

We cannot claim that any particular set of AFPI experiments will find *all* failure propagation paths, e.g., because we depend on the applied workload to exercise all affected components. However, when using AFPI in recovery via recursive microreboots, we can view the use of the f-map as an optimization that impacts recovery performance but not recovery correctness. If the failure had been non-transient, then neither full recovery nor partial recovery guided by the f-map would have cured it.

We cannot say that the f-maps converge after the fault injection experiments have completed and we don't know how many fault paths remain undiscovered. Preliminary evidence suggests that the type of experiments we've run are sufficient, but there is no proof that an exhaustive injection of the application-visible and system-generated faults would achieve this kind of coverage. We would expect to experience rapidly diminishing returns in trying to perform exhaustive injection.

A.3 Summary

We focused on applying Automated Failure-Path Inference to applications built on Java 2 Enterprise Edition middleware, because such applications tend to be highly modular and rely on a well-defined set of runtime services whose implementation we can change to add fault injection and monitoring behaviors.

The experiments we described show that AFPI automatically and dynamically generates f-maps that find runtime dependencies that static call graph analysis might miss. AFPI-generated f-maps

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

correctly omit dependencies that appear in the static call graph but do not result in observed fault propagation at runtime. The dynamically generated f-maps can capture dependency information per fault type, providing higher resolution than many static techniques.

Injecting Java exceptions to represent real operational faults is reasonable, and in particular, certain common classes of application-generic faults (such as resource exhaustion) are often manifest as JVM exceptions. While injecting correlated faults, we did not find any new f-map edges, but expect this to be due to the simple applications, rather than to the AFPI implementation. For similar reasons, we have not yet investigated in depth the stability of f-maps in the presence of quasi-deterministic faults, such as those introduced by pernicious firmware bugs.

AFPI's staging phase took about three hours for a nontrivial application consisting of over 26K lines of code and required no manual inspection or knowledge of the application itself; the additional overhead of leaving AFPI in place during production operation was negligible. Although AFPI may not capture all propagation paths, we aim for a solution that is the right tradeoff between complexity/difficulty/cost and effectiveness.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Appendix B

Roadmap to Representative Publications

Microreboot-based recovery and the related properties of crash-only software systems were introduced in [CF01], as described in chapter 5, and in [CF03], as covered by chapter 4.

The initial forays into reboot-based recovery, described in chapter 2, appeared in [Can00] and [CCF+02]; an expanded version of the latter was published as [CCF04].

An extensive exploration of microreboots in the eBid/JBoss J2EE system appeared in [CKF+04]; this work provided material for chapters 7 and 8. Some of the ramifications of microreboot-based recovery, described in chapters 8 and 10, appeared in [CKK+03] and [CKKF04] (autonomous recovery and fault detection/localization) and in [CBFP04] (multi-tier recovery). Some additional work on detection of application-level failures appeared in [BFB+05]. The contents of appendix A are largely based on [CDCF03].

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

Bibliography

[ABB+86] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proc. USENIX Summer Conference*, Atlanta, GA, 1986.

[ABC+02] Atul Adya, William Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John Douceur, Jon Howell, Jacob Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[Ach02] Anurag Acharya. Reliability on the cheap: How I learned to stop worrying and love cheap PCs. In *2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002. Invited Talk.

[Ada84] E. Adams. Optimizing preventative service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[ADAD01a] Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, 2001.

[ADAD01b] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau, Germany, 2001.

[AIS+01] Tony Adams, Rober Igou, Ron Silliman, Alan Mac Neela, and Eric Rocco. Sustainable infrastructures: How IT services can address the realities of unplanned downtime. Research Brief 97843a, Gartner Research, May 2001. Strategy, Trends & Tactics Series.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[AK76] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *Proc. 2nd International Conference on Software Engineering*, San Francisco, CA, 1976.

[Are02] Luis Arellano. Efinance, inc. Personal Comm., 2002.

[Bar81] J. F. Bartlett. A NonStop kernel. In *Proc. 8th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1981.

[Bar02] Wendy Bartlett. HP NonStop server: Overview of an integrated architecture for fault tolerance. In *2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, Oct 2002. Invited Talk.

[Bar04] Michael Barnes. J2EE application servers: Market overview. The Meta Group, March 2004.

[BBK00] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. In *Proc. 9th International WWW Conference*, Amsterdam, Holland, 2000.

[BDGR97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. DISCO: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[BFB+05] Peter Bodik, Greg Friedman, Luke Biewald, H.T. Levine, and George Candea. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proc. 2nd IEEE International Conference on Autonomic Computing*, Seattle, WA, June 2005.

[BP03] Aaron B. Brown and David A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proc. USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[Bre00] Eric Brewer. Inktomi insights. Personal Comm., 2000.

[Bre01a] Eric Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[Bre01b] Eric Brewer. Running Inktomi. Personal Comm., 2001.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[Bro95] Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, Anniversary edition, 1995.

[BS92] Mary Baker and Mark Sullivan. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. In *Proc. Summer USENIX Technical Conference*, San Antonio, TX, 1992.

[BS01] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proc. IEEE High-Assurance System Engineering Symposium*, Boca Raton, FL, 2001.

[BST02] Peter A. Broadwell, Naveen Sastry, and Jonathan Traupman. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, New York, NY, 2002.

[CAK+04] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Eric A. Brewer, David Patterson, and Armando Fox. Path-based failure and evolution management. In *Proc. 1st Symposium on Networked Systems Design and Implementation*, San Francisco, CA, 2004.

[Can00] George Candea. Medusa: A platform for highly available execution. CS244C (Distributed Systems) course project, Stanford University, `http://stanford.edu/~candea/papers/medusa`, June 2000.

[CBFP04] George Candea, Aaron Brown, Armando Fox, and David Patterson. Recovery-oriented computing: Building multi-tier dependability. *IEEE Computer*, November 2004.

[CC00] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? A case study using open-source software. In *Proc. International Conference on Dependable Systems and Networks*, New York, NY, 2000.

[CC02] Carl Claunch and Jim Cassell. High-availability study: Observations and key conclusions. Technical report, Gartner Research, October 8 2002.

[CCF+02] George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, and Rakesh Gowda. Reducing recovery time in a small recursively restartable system. In *Proc. International Conference on Dependable Systems and Networks*, Washington, DC, June 2002.

[CCF04] George Candea, James Cutler, and Armando Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal*, 56(1-4):213–238, March 2004.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[CD01] Grzegorz Czajkowski and Laurent Daynés. Multitasking without compromise: A virtual machine evolution. In *Proc. Conference on Object Oriented Programming Systems Languages and Applications*, Tampa Bay, FL, 2001.

[CDCF03] George Candea, Mauricio Delgado, Michael Chen, and Armando Fox. Automatic failure-path inference: A generic introspection technique for software systems. In *Proc. 3rd IEEE Workshop on Internet Applications*, San Jose, CA, 2003.

[CF01] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau, Germany, 2001.

[CF03] George Candea and Armando Fox. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, 2003.

[Cho97] Timothy C.K. Chou. Beyond fault tolerance. *IEEE Computer*, 30(4):47–49, 1997.

[Cho03] Timothy C.K. Chou. Personal communication. Oracle Corp., 2003.

[CJ02] Howard Cohen and Ken Jacobs. Personal comm. Oracle, 2002.

[CKF+02] Mike Chen, Emre Kiciman, Eugene Fratkin, Eric Brewer, and Armando Fox. Pinpoint: Problem determination in large, dynamic, Internet services. In *Proc. International Conference on Dependable Systems and Networks*, Washington, DC, 2002.

[CKF+04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – a technique for cheap recovery. In *Proc. 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[CKK+03] George Candea, Pedram Keyani, Emre Kiciman, Steve Zhang, and Armando Fox. JAGR: An autonomous self-recovering application server. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, June 2003.

[CKKF04] George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, to appear 2004.

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 1999.

[CMB+01] Deqing Chen, Alan Messer, Philippe Bernadat, Guangrui Fu, Zoran Dimitrijevic, David Jeun Fung Lie, Durga Mannaru, Alma Riska, and Dejan Milojicic. JVM susceptibility to memory errors. In *Submitted to Java Virtual Machine Research and Technology Symposium*, 2001.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[CMZ02] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *Proc. 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, 2002.

[CNFC04] Keith Coleman, Jim Norris, Armando Fox, and George Candea. OnCall: Defeating spikes with a free-market server cluster. In *Proc. International Conference on Autonomic Computing*, New York, NY, May 2004.

[CNRA96] Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani, and Christopher M. Aycock. The Rio file cache: Surviving operating system crashes. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, 1996.

[Com02] Computer_Associates. Unicenter CA-SYSVIEW realtime performance management. `http://www.ca.com`, October 2002.

[Cou01] Patrick Cousot, editor. *Static Analysis*. Springer Verlag, 2001.

[CR72] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21(6):546–556, June 1972.

[CRD+95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, 1995.

[CV98] Judith Crow and Ben Di Vito. Formalizing space shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.

[CYC+01] Andy Chou, Jun-Feng Yang, Bejamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Lake Louise, Canada, 2001.

[CZL+04] Michael Chen, Alice Zheng, Jim Lloyd, Michael Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Proc. Intl. Conference on Autonomic Computing*, New York, NY, 2004.

[DEF+96] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, P. Sharma, and A. Helmy. Protocol independent multicast (PIM), sparse mode protocol: Specification, March 1996. Internet Draft.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[DMM04] Ada Diaconescu, Adrian Mos, and John Murphy. Automatic performance management in component based software systems. In *First International Conference on Autonomic Computing*, New York, NY, May 2004.

[DN66] Ole-Johan Dahl and Kristen Nygaard. Simula—an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, Sep 1966.

[Duv04] Sreeram Duvur. Personal comm. Sun Microsystems, 2004.

[EBa04] Information obtained under an agreement that prohibits disclosure of the company's name, May 2004.

[Enr02] Patricia Enriquez. Failure analysis of the pstn, January 2002. http://roc.cs.berkeley.edu/retreats/spring_02/d1_slides/pdf/RocTalk.pdf.

[Fea03] C. Fu and R. Martin et al. Compiler-directed program-fault coverage for highly available internet applications. Technical report, Rutgers University Computer Science Dept., 2003.

[Fet03] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, February 2003.

[FGC+97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Maló, France, 1997.

[Fis97] Charles Fishman. They write the right stuff. *FastCompany*, Jan 1997.

[FJ94] Sally Floyd and Van Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking*, 2(2):122–136, April 1994.

[FJLM95] Sally Floyd, Van Jacobson, C. Liu, and Steven McCanne. A reliable multicast framework for light-weight sessions and application level framing. In *Proc. ACM SIGCOMM Conference*, Boston, MA, 1995.

[FP02] Armando Fox and David Patterson. When does fast recovery trump high reliability? In *Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002.

[GBHC00] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for Internet service construction. In *Proc. 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[GC89] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, 1989.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[GHKT96] S. Garg, Y. Huang, C. Kintala, and K.S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. In *Proc. ACM Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, 1996.

[GMF+99] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Internet RFC 2616, June 1999.

[GMVT98] Sachin Garg, Aad Van Moorsel, K. Vaidyanathan, and Kishor S. Trivedi. A methodology for detection and estimation of software aging. In *Proc. 9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998.

[Gol74] Robert. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, June 1974.

[GPTT95] S. Garg, A. Puliafito, M. Telek, and K.S. Trivedi. Analysis of software rejuvenation using Markov regenerative stochastic Petri nets. In *Proc. 6th International Symposium on Software Reliability Engineering*, Toulouse, France, 1995.

[GR93] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[Gra78] Jim Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, J. H. Saltzer, and G. Seegmüller, editors, *Operating Systems, An Advanced Course*, volume 60, pages 393–481. Springer, 1978.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proc. International Conference on Very Large Data Bases*, Cannes, France, 1981.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.

[Gra90] Jim Gray. A census of tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, Oct 1990.

[Gri01] Steven D. Gribble. Robustness in complex systems. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[Hew02] Hewlett_Packard. Integrated and correlated enterprise management with the open management interface specification. HP OpenView whitepaper, `http://www.openview.hp.com`, 2002.

[HF03] Andrew C. Huang and Armando Fox. Decoupled storage: State with stateless-like properties. Submitted to the 22nd Symposium on Reliable Distributed Systems, 2003.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Maló, France, 1997.

[HK93] Yennun Huang and Chandra M. R. Kintala. Software implemented fault tolerance: Technologies and experience. In *Proc. 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France, 1993.

[HKKF95] Yennun Huang, Chandra M. R. Kintala, Nick Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, 1995.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proc. Winter USENIX Technical Conference*, San Francisco, CA, 1994.

[HMC88] Rober Haskin, Yoni Malachi, and Gregory Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[HvE02] Chris Hawblitzel and Thorsten von Eicken. Luna: A flexible Java protection system. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[Int01] International_Business_Machines. IBM director software rejuvenation. White Paper, Jan 2001.

[Int02] International_Business_Machines. Tivoli monitoring resource model reference. Document Number SH19-4570-01, `http://www.tivoli.com`, 2002.

[Jac03] Dean Jacobs. Distributed computing with BEA WebLogic server. In *Proc. Conference on Innovative Data Systems Research*, Asilomar, CA, 2003.

[JBo02] JBoss. Homepage. http://www.jboss.org/, 2002.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer Magazine*, Jan 2003.

[Kem98] Robert W. Kembel. *The Fibre Channel Consultant: A Comprehensive Introduction*. Northwest Learning Associates, 1998. page 8.

[Key] Keynote Systems. http://www.keynote.com/.

[KF05] Emre Kiciman and Armando Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, September 2005.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[KIBW99] Zbigniew T. Kalbarczyk, Ravi K. Iyer, S. Bagchi, and Keith Whisnant. Chameleon: a software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10:560–579, 1999.

[KIR+99] Zbigniew Kalbarczyk, Ravishankar K. Iyer, G.L. Ries, J.U. Patel, M.S. Lee, and Y. Xiao. Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. *IEEE Transactions on Software Engineering*, 25(5):619–632, September/October 1999.

[KPGV03] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. In *Proc. 5th Annual International Active Middleware Workshop*, June 2003.

[Kuh97] D. R. Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer*, 30(4):31–36, Apr 1997.

[Lap91] Jean-Claude Laprie, editor. *Dependability: Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems. Springer Verlag, Dec 1991.

[LC97] David E. Lowell and Peter M. Chen. Free transactions with Rio Vista. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Maló, France, 1997.

[LCC00] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proc. 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000.

[LCD+03] E. Lassettre, D. Coleman, Y. Diao, S. Froelich, J. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L. Russell, M. Surendra, V. Tseng, N. Wadia, and P. Ye. Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges with Resource Actions that have Lead Times. In *Proc. of 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, San Diego, CA, June 2003.

[LeF01] William LeFebvre. CNN.com–Facing a world crisis. Talk at *15th USENIX Systems Administration Conference*, 2001.

[Lev03] H.T. Levine. Personal communication. EBates.com, 2003.

[LF03] Ben Ling and Armando Fox. A self-tuning, self-protecting, self-healing session state management layer. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, 2003.

[LGWJ01] Tirthankar Lahiri, Amit Ganesh, Ron Weiss, and Ashok Joshi. Fast-Start: Quick fault recovery in Oracle. In *Proc. ACM International Conference on Management of Data*, Santa Barbara, CA, 2001.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[LKF04] Benjamin Ling, Emre Kiciman, and Armando Fox. Session state: Beyond soft state. In *Proc. 1st Symposium on Networked Systems Design and Implementation*, San Francisco, CA, 2004.

[Lyu95] Michael R. Lyu. *Handbook of Software Reliability Engineering*. McGraw Hill, 1995.

[Man01] Udi Manber. Personal communication, 2001.

[McC59] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. In John McCarthy and Marvin L. Minsky, editors, *Artificial Intelligence. Quarterly Progress Report No. 53*. MIT Research Lab of Electronics, Cambridge, MA, April 1959.

[MD99] Brendan Murphy and Neil Davies. System reliability and availability drivers of Tru64 UNIX. In *Proc. 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, 1999. Tutorial.

[Mes04a] Greg Messer. Personal communication. US Bancorp, 2004.

[Mes04b] Adam Messinger. Personal comm. BEA Systems, 2004.

[Mey80] J. F. Meyer. On evaluating the performability of degradable computer systems. *IEEE Transactions on Computers*, C-29:720–731, Aug 1980.

[MG95] Brendan Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering Intl.*, 11:341–353, 1995.

[Mil68] R.B. Miller. Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference*, volume 33, 1968.

[Mit04] Nick Mitchell. IBM Research. Personal Comm., 2004.

[MMS+00] Dejan Milojicic, Alan Messer, James Shau, Guangrui Fu, and Alberto Munoz. Increasing relevance of memory hardware errors. a case for recoverable programming models. In *Proc. ACM SIGOPS European Workshop*, Kolding, Denmark, 2000.

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. 11th ACM Symposium on Operating Systems Principles*, Austin, TX, 1987.

[MS00] Evan Marcus and Hal Stern. *Blueprints for High Availability*. John Wiley & Sons, Inc., New York, NY, 2000.

[MS03] Nick Mitchell and Gary Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. 17th European Conf. on Object-Oriented Programming*, Darmstadt, Germany, 2003.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[Nat98] The investigation of flight KAL-801, b-747-300, 1997-08-06. National Transportation Safety Board, March 26 1998.

[NBMN02] Kiran Nagaraja, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Using fault model enforcement to improve availability. In *Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002.

[NIS] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing, Gaithersburg, MD, May 2002.

[NOC02] NOCPulse. Command center overview. `http://nocpulse.com`, 2002.

[OBS99] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer USENIX Technical Conference*, Monterey, CA, June 1999.

[Off92] U.S. General Accounting Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report of the U.S. General Accounting Office, GAO/IMTEC-92-26, GAO, 1992.

[OGP03] David Oppenheimer, Archana Ganapathi, and David Patterson. Why do Internet services fail, and what can be done about it? In *Proc. 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, 2003.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pal02] Anil Pal. Personal communication. Yahoo!, Inc., 2002.

[Pou04] Kevin Poulsen. Software bug contributed to blackout. http://www.securityfocus.com/news/8016, February 2004. SecurityFocus.

[RCD+04] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proc. 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[Ree98] Glenn Reeves. What really happened on Mars? RISKS-19.49, January 1998.

[Rei04] Darrell Reimer. IBM Research. Personal comm., 2004.

[Res02] Resonate. Application performance management for business critical applications. `http://resonate.com`, 2002.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[RLC01] Rodrigo Rodrigues, Barbara Liskov, and Miguel Castro. BASE: Using abstraction to improve fault tolerance. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, 2001.

[RO91] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1991.

[RUB] RUBiS project web page. http://rubis.objectweb.org/.

[SABL04] Michael Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proc. 6th ACM/USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proc. 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003.

[SC91] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability – a study of field failures in operating systems. In *Proc. 21st International Symposium on Fault-Tolerant Computing*, Montréal, Canada, 1991.

[Sco99] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford, CT, March 19 1999. PRISM for Enterprise Operations.

[SDLS02] Pete Soper, Peter Donald, Doug Lea, and Miles Sabin. Application isolation API specification. Java Specification Request No. 121, `http://jcp.org/en/jsr/detail?id=121`, 2002.

[SG99] Lisa Spainhower and Thomas A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5/6), 1999.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the USENIX Summer Conference*, pages 119–130, Portland, OR, 1985.

[SM00] G. Gordon Schulmeyer and Garth R. MacKenzie. *Verification and Validation of Modern Software-Intensive Systems*. Prentice Hall, Englewood Cliffs, NJ, 2000.

[Smi02] Wayne D. Smith. TPC-W: Benchmarking an E-Commerce solution. Transaction Processing Council, 2002.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[Sto87] Michael Stonebraker. The design of the Postgres storage system. In *Proc. 13th Conference on Very Large Databases*, Brighton, England, 1987.

[Sub02] Shanti Subramanyam. ECperf benchmark specification. Java Specification Request No. 4, `http://jcp.org/en/jsr/detail?id=004`, 2002.

[Sun] Sun Microsystems. http://java.sun.com/j2ee/.

[Sun02] Sun_Microsystems. Java Pet Store Demo. http://developer.java.sun.com/developer/releases/petstore/, 2002.

[THL01] Patrick Tullmann, Mike Hibler, and Jay Lepreau. Janos: A Java-oriented OS for active networks. *IEEE Journal on Selected Areas in Communications*, 19(3):501–510, 2001.

[Tio04] Tiobe programming community index. http://www.tiobe.com/tiobe_index/, November 2004.

[TJWR99] Bill Tuthill, Karen Johnson, Susan Wilkening, and Dean Roe. *IRIX Checkpoint and Restart Operation Guide*. Silicon Graphics, Inc., Mountain View, CA, 1999.

[Vor03] Lisa Vorderbrueggen. Glitch gives BART commuters free ride. *Contra Costa Times*, December(18), 2003. http://www.bayarea.com/mld/cctimes/content_syndication/local_news/7519819.htm.

[WBS+98] K. Whisnant, S. Bagchi, B. Srinivasan, Z. Kalbarczyk, and R.K. Iyer. Incorporating reconfigurability, error detection, and recovery into the Chameleon ARMOR architecture. Technical Report CRHC-98-13, University of Illinois at Urbana-Champaign, 1998.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, 2001.

[WHV+95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra M. R. Kintala. Checkpointing and its applications. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, 1995.

[WIH+02] K. Whisnant, RK Iyer, P. Hones, R. Some, and D. Rennels. Experimental evaluation of the REE SIFT environment for spaceborne applications. In *Proc. International Conference on Dependable Systems and Networks*, Washington, DC, 2002.

[Wil82] R. J. Willett. Design of recovery strategies for a fault-tolerant no. 4 electronic switching system. *The Bell System Technical Journal*, 61(10):3019–3040, Dec 1982.

[Wir88] Niklaus Wirth. The programming language Oberon. *Software—Practice and Experience*, 18(7):671–690, 1988.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.

[WKI00] K. Whisnant, Z. Kalbarczyk, and R.K. Iyer. Micro-checkpointing: Checkpointing for multi-threaded applications. In *Proc. IEEE Intl. On-Line Testing Workshop*, 2000.

[Woo95] Alan Wood. Predicting client/server availability. *IEEE Computer*, 28(4):41–48, April 1995.

[Woo03] Alan P. Wood. Software reliability from the customer view. *IEEE Computer*, 36(8):37–42, August 2003.

[WS91] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steve Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[ZDE+93] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Network*, 7(5), September 1993.

[Zon01] Zona research bulletin: The need for speed II, April 2001.

This PDF was built from a very old LaTeX repository, so parts of the thesis might be missing or not display properly.