DEPENDABILITY BENCHMARKING FOR COMPUTER SYSTEMS

Edited by

Karama Kanoun
Lisa Spainhower

A JOHN WILEY & SONS, INC., PUBLICATION

DEPENDABILITY BENCHMARKING FOR COMPUTER SYSTEMS

IEEE Computer Society Publications

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the CS Store at *http://computer.org/cspress* for a list of products.

IEEE Computer Society / Wiley Partnership

The IEEE Computer Society and Wiley partnership allows the CS Press authored book program to produce a number of exciting new titles in areas of computer science and engineering with a special focus on software engineering. IEEE Computer Society members continue to receive a 15% discount on these titles when purchased through Wiley or at wiley.com/ieeecs.

To submit questions about the program or send proposals e-mail j.wilson@computer.org. Telephone +1-714-821-8380. **Additional information regarding the Computer Society authored book program can also be accessed from our web site at** *http://computer.org/cspress.*

DEPENDABILITY BENCHMARKING FOR COMPUTER SYSTEMS

Edited by

Karama Kanoun
Lisa Spainhower

A JOHN WILEY & SONS, INC., PUBLICATION

Trademarks

IBM, Rational, and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

TPC and TPC-C are trademarks of the Transaction Processing Performance Council.

SPEC and SPECjAppServer 2004 are registered trademarks of the Standard Performance Evaluation Corporation.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data is available.

ISBN 978-0470-23055-8

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

CONTENTS

PREFACE

The book is written for engineers and researchers working in the field of dependability and, more generally, in the field of system engineering. The book could be adopted as an advanced undergraduate, graduate, or continuing education textbook. The variety of benchmarks presented provide useful starting points for further research in dependability benchmarking.

This book consists of sixteen chapters, prepared by researchers and engineers working in the dependability field for several years and, more specifically, on dependability benchmarking during the more recent years. These chapters illustrate the current multiplicity of approaches to dependability benchmarking, and the diversity of benchmark measures that can be evaluated.

Chapters 1 through 6 examine system-level benchmarks that focus on various aspects of dependability using different measurement methods. Chapters 7 to 16 focus on benchmarks for system components: control algorithms (Chapter 7), intrusion detectors (Chapter 8), fault-tolerance algorithms (Chapter 9), operating systems (Chapters 10 to 15), and microprocessors (Chapter 16).

Chapter 1 describes an autonomic computing benchmark that measures system resiliency. In common with other benchmarks, it has a quantitative throughput metric to capture the impact of disturbances. The autonomic computing benchmark also has a qualitative metric representing the level of human interaction needed to detect, analyze, and recover from disturbances.

Chapter 2 contains three benchmarks that analytically measure reliability, availability, and serviceability of a system. All three are intended to allow a consistent dependability feature versus cost analysis for widely varying system architectures.

Chapter 3 covers two quantitative benchmarks that concentrate on the recovery of standalone systems and enterprise clusters. Recovery is broadly defined to be the time required for a system to return to service. For each benchmark, a scoring and weighting methodology is described.

The benchmark in Chapter 4 changes the physical environment around the system in order to quantitatively measure susceptibility to silent data corruption.

Chapters 5 and 6 include performance measurements in the presence of faults, providing performance degradation (due to faults) and availability measures to end users of on-line transaction processing and Web serving systems. Chapter 5 discusses a benchmark for on-line transaction processing that focuses on availability using direct experimental measures. Chapter 6 includes a measurement-based benchmark for Web servers that encompasses availability, accuracy, resilience, and service degradation.

Chapter 7 is dedicated to the benchmarking of automotive control applications running inside engine control units. It addresses the safety of the software applications from the perspective of the engine they control. It is aimed at supporting the selection or the purchase of automotive applications for electronic controller units.

In Chapters 8 and 9, the systems under benchmarks are, respectively, the intrusion-detection mechanisms and fault-tolerance algorithms.

Chapter 8 discusses issues that should be addressed when benchmarking intrusion detectors in the cyber domain. It is worth mentioning that no intrusion detector benchmark is currently available, and that this chapter does not develop a benchmark per se.

Chapter 9 addresses Byzantine protocols dedicated to Byzantine-fault tolerance. The benchmark is aimed at assessing the effectiveness of Byzantine-fault tolerance implementation. It is illustrated by a specific protocol, the Castro–Liskov protocol.

Chapters 10–15 exemplify the multifaceted feature of dependability benchmarking, illustrated by operating systems. The benchmarks presented differ mainly in the nature of the faults considered: Chapter 10 addresses internal faults, whereas chapters 11–15 are concerned with external faults. Chapters 11, 12, and 13 address faults in the software application, Chapter 14 addresses faults in device drivers, and Chapter 15 addresses faults in hardware platforms.

Chapter 10 defines a benchmark that captures the user's expectations of product reliability during common user experiences. Although the benchmark can be applied to any software product, this chapter focuses on the application of this benchmark for operating-systems development. Its aim is to help the operating-system developer to improve, during its development, the operating system's reliability.

Chapters 11–14 present benchmarks based on experimentation. They consider the operating system as a black box, examine it only through its available inputs, and observe its behavior only through the operating-system outputs. The ultimate objective of end-user's benchmarks is to improve the application software, the device drivers, or the hardware platform. However, the benchmark results may indicate weaknesses in the operating system that the operating-system developer may use, with further analyses, to improve the operating system.

Chapter 11 is dedicated to the Ballista project, the first operating-system robustness-testing approach. It reexamines the design decisions made and the lessons learned from a decade of robustness-testing research.

Chapter 12 elaborates on the Ballista approach and develops a benchmark evaluating, in addition to the operating-system robustness, two measures: the system response time and restart time in the presence of faults.

Chapter 13 focuses on real-time kernels. It characterizes the response time predictability, based on the divergence of the response time in the presence of fault, and the frequency of out-of-boundary responses times.

Chapter 14 concentrates on failures modes of the operating systems induced by faulty drivers. Additionally, it evaluates three complementary measures: responsiveness, availability, and workload safety.

Chapter 15 considers in a first step the operating system as a black box and develops an end-user benchmark, then shows how this benchmark can be complemented to help the operating-system developer improve the operating system's dependability.

Chapter 16 is dedicated to microprocessor benchmarks with respect to electrical charge induced by high-energy particles, referred to as soft errors. As the benchmark presented does not require any specific knowledge about the microprocessors, it can be used by the manufacturers or end users.

ACKNOWLEDGMENTS

Beyond the acknowledgements contained in the chapters of this book, we would like to especially thank:

- The European Commission, which has partially financed the work achieved within the DBench project, and, particularly, Angelo Marino, the scientific officer of the project, for his continual support and advice;
- IFIP, the International Federation for Information Processing, and, particularly, the 10.4 Working Group on Dependable Computing and Fault Tolerance, which created the Special Interest Group on Dependability Benchmarking (SIGDeB). In particular, Jean Arlat was the Working Group Chair who fostered the SIGDeB's creation;
- Carnegie Mellon University, IBM, LAAS-CNRS, and Sun Microsystems, who provided support for the SIGDeB meetings; and
- All SIGDeB members and DBench partners, who contributed to preparing the way for the benchmarks presented in the book. Particularly active members beyond the authors represented in this book include: Wendy Bartlett (Hewlett Packard), Jean-Paul Blanquart (EADS-Astrium), Mario Dal Cin (retired from Friedrich Alexander University, Erlangen-Nürnberg), Jean-Claude Laprie (LAAS-CNRS), Brendan Murphy (Microsoft), and Don Wilson (retired from Hewlett Packard).

Finally, we would like to acknowledge Jim Gray, who inspired work in this area and actively nurtured both research groups and individuals in this field of research.

KARAMA KANOUN
LISA SPAINHOWER

Toulouse, France
Poughkeepsie, New York
May 2008

CONTRIBUTORS

Arnaud Albinet, LAAS-CNRS, University of Toulouse, Toulouse, France
Jean Arlat, LAAS-CNRS, University of Toulouse, Toulouse, France
Ricardo Barbosa, Critical Software, S. A., Coimbra, Portugal
William Bryson, Sun Microsystems, Inc., Santa Clara, California,
Joyce Coleman, IBM Toronto Laboratory, Markham, Ontario, Canada
Cristian Constantinescu, AMD, Fort Collins, Colorado
Diamantino Costa, Critical Software, S. A., Coimbra, Portugal
Yves Crouzet, LAAS-CNRS, University of Toulouse, Toulouse, France
David de-Andres, ITACA, Technical University of Valencia, Valencia, Spain
John DeVale, Applied Physics Laboratory, Johns Hopkins University, Laurel, Maryland
Kobey DeVale, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania
João Durães, ISEC/CISUC, University of Coimbra, Coimbra, Portugal
Richard Elling, Sun Microsystems, Inc., San Diego, California
Jean-Charles Fabre, LAAS-CNRS, University of Toulouse, Toulouse, France
Mario R. Garzia, Microsoft Corporation, Redmond, Washington
Pedro Gil, ITACA, Technical University of Valencia, Valencia, Spain
Weining Gu, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois
Ravishankar Iyer, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois
Ali Kalakech, LAAS-CNRS, University of Toulouse, Toulouse, France
Zbigniew Kalbarczyk, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois
Karama Kanoun, LAAS-CNRS, University of Toulouse, Toulouse, France
Philip Koopman, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania
Tony Lau, IBM Toronto Laboratory, Markham, Ontario, Canada
Bhushan Lokhande, IBM Rochester Laboratory, Rochester, Minnesota
Henrique Madeira, DEI/CISUC, University of Coimbra, Coimbra, Portugal
James Mauro, Sun Microsystems, Inc., Somerset, New Jersey

Ricardo Maia, Critical Software, S. A., Coimbra, Portugal

Roy A. Maxion, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania

Francisco Moreira, Critical Software, S. A., Coimbra, Portugal

Priya Narasimhan, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, Pennsylvania

Ira Pramanick, Sun Microsystems, Inc., Menlo Park, California

Juan-Carlos Ruiz, ITACA, Technical University of Valencia, Valencia, Spain

Ana-Elena Rugina, LAAS-CNRS, University of Toulouse, Toulouse, France

Peter Shum, IBM Toronto Laboratory, Markham, Ontario, Canada

Lisa Spainhower, IBM, Poughkeepsie, New York

Dong Tang, Sun Microsystems, Inc., Santa Clara, California

Kymie M. C. Tan, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania

Marco Vieira, DEI/CISUC, University of Coimbra, Coimbra, Portugal

Robert Wisniewski, IBM Rochester Laboratory, Rochester, Minnesota

Sonya J. Wierman, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, Pennsylvania

Mary Peterson Yost, IBM, Software Group, Somers, New York

Pedro Yuste, ITACA, Technical University of Valencia, Valencia, Spain

PROLOGUE DEPENDABILITY BENCHMARKING: A REALITY OR A DREAM?

Karama Kanoun, Philip Koopman, Henrique Madeira, and Lisa Spainhower

Until recently, for computer systems, a benchmark referred implicitly to a performance benchmark. For example, the seminal handbook on database and transactional systems benchmarking, published in the early 1990s [Gray 1993], was totally dedicated to performance benchmarks. The handbook title does not even mention "performance" benchmarking. De facto, a benchmark has been a performance benchmark.

Many factors converge to raise the importance of dependability of today's computer-based systems. These include global 24/7 operations, on-line businesses, and the increasing complexity of systems. At the same time, high levels of performance are available at rapidly decreasing cost, decreasing the predominance of absolute performance and relative price/performance as indicators of customer satisfaction. Dependability is more and more playing a determinant role. However, although industry standard benchmarks are easily and widely used to measure computer performance in a deterministic and reproducible manner, dependability benchmarking is in its infancy. Nonetheless, although far from the maturity of current performance benchmarks, dependability benchmarking is making definite progress, as demonstrated by efforts in industry and academia illustrated in this book.

A dependability benchmark is intended to characterize system behavior in the presence of faults that could be internal or external to the system being benchmarked. Potential faults include component failures, hardware or software design flaws, faults in other systems interacting with the benchmarked systems, operator errors, and perturbations in the environment. Benchmarking the dependability of a system consists of evaluating dependability or dependability-and-performance-related measures in the presence of faults, in a well-structured and standardized way. Measures may characterize the system in a com-

prehensive way; that is, they may address the service delivery level and take into account the occurrence of various events impacting its behavior and their consequences. On the other hand, they may characterize specific features of the system such as coverage provided by fault tolerance mechanisms, time to restart the system, or time to system backup. As a consequence, numerous benchmarking measures are of interest.

The key aspect that distinguishes benchmarking from existing evaluation and validation techniques is that a benchmark fundamentally represents an agreement (explicit or tacit) that is accepted by those who make and sell computers and those who purchase them. This technical agreement states the measures, the way the measures are obtained, and the domain (e.g., application area) in which these measures are considered valid and meaningful. In other words, a real benchmark is something that the user community and the computer industry accept as representative enough of a given application domain to be deemed useful, and to be generally used as a way of measuring specific features of a computer system and, consequently, a way to compare different systems.

Currently, several organizations and research groups are carrying out promising work on dependability benchmarking, and, as a result, a great variety of benchmarks has been defined and implemented in the last decade. Many of these dependability benchmarks resulted from work performed by single institutions (proprietary benchmarks defined in response to internal needs, or work on particular aspects of dependability benchmarking). A few other benchmarks resulted from coordinated and concerted work between several institutions. Even though standard, well established, and widely agreed on dependability benchmarks, approved by recognized consortiums as in the case of performance benchmarks, do not really exist, the dependability benchmarks developed so far are paving the way for such standard benchmarks.

The results of a dependability benchmark are aimed at either characterizing system dependability capabilities in a qualitative manner (e.g., on the basis of the dependability features supported or claimed, such as the on-line error detection, fail-silent failure mode), or quantitatively assessing these properties. Dependability benchmark results can be useful for both the end users and vendors to:

- Characterize the dependability of a component or a system, qualitatively or quantitatively
- Track dependability evolution for successive versions of a product
- Identify weak parts of a system, requiring more attention and perhaps needing some improvements by tuning a component to enhance its dependability, or by tuning the system architecture (e.g., adding fault tolerance) to ensure a suitable dependability level
- Compare the dependability of alternative or competitive solutions according to one or several dependability attributes

A dependability benchmark can be performed in various phases of the system life cycle. The measures obtained for a specific phase are then helpful for the current or subsequent phases. The purpose of a benchmark may vary significantly along the system life cycle. For example:

- During the very early design phases, a dependability benchmark could support the decision whether to purchase a particular hardware or software component or platform that is to be integrated into a particular application.

- During development, results could be used to reveal weak points and to monitor the improvement actually achieved by fault removal activities (e.g., regression testing).
- For operational life, benchmark results could be useful to evaluate the impact of faults (hardware, software, or operator faults) on system dependability.

With many current systems, the traditional role of dependability assessment methods in the development life cycle of computing systems and applications (i.e., measuring used as an inherent step of the improvement process, and as a verification and validation facilitator) must be expanded in order to address the technical problems resulting from current component-based development practices involving intensive reuse of components. The use of dependability benchmarks is an important step in that direction. In fact, it is now a common practice for large-scale software development to reuse preexisting ("off-the-shelf") components (normally, general-purpose components possibly from the open-source community or components easily adapted to the requirements of the new system) and to develop from scratch only such further components and "glue" code as are also needed. Given the high costs of designing and implementing new software, vendors see the reuse of components as a way to reduce development effort and to achieve rapid time to market. However, component reuse introduces unknown risks of failure, as the new operational conditions may differ substantially from those that the components were initially designed for, and the new operational conditions may cause the activation of unknown residual faults or produce new component interaction faults. The use of dependability benchmarking in the development/integration of such composite systems seems very useful for component selection and to assess dependability measures of the whole system.

The benchmark performer (i.e., the person or entity actually performing the benchmark) can be a system manufacturer (or vendor), a system integrator, a third party, or an end user. These entities have different visions of the target system and, as a consequence, they have diverse expectations of the benchmark results. For example, system vendors have low-level access and observation points, whereas other entities usually can only make use of available inputs and outputs to interact with the system and observe its behavior to assess its dependability.

To sum up, dependability benchmarks allow objective characterization of system dependability. Therefore, they can provide a good means for fair comparison between alternative systems. They can also be used for guiding development efforts of system providers, and for supporting acquisition choices of system purchasers, or for comparing the dependability of new versions of a system with respect to previous ones.

Due to the above various aspects of dependability benchmarks, a range of approaches for dependability benchmarking has been followed. This book reflects these varieties. Some of them address the end-user points of view and others address primarily the vendor point of view (even though the user point of view is always there). Additionally, these benchmarks are at different maturity stages.

Benchmarking a system or a component is typically based on experimentation, or modeling, or on both. Experimental results may be obtained from controlled experiments defined in the benchmark or from the observation of the system in the field, under its operational conditions. Controlled experimentation consists of applying a workload and a faultload to the system being benchmarked to obtain the benchmark measures; they are based on fault injection techniques. All the benchmarks presented in this book can be understood in terms of the reference model for implementing dependability benchmarks illustrated in Figure 1. However, some work focuses mainly on the controlled experimenta-

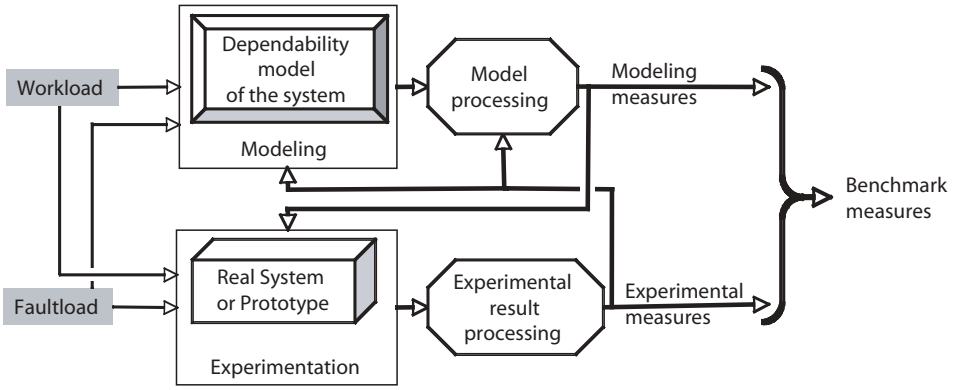

Figure 1. Reference model for implementing dependability benchmarks.

tion parts (putting effort into the selection of the workload, faultload, and experimental measures), some work gives more importance to modeling, whereas other work emphasizes obtaining experimental results from system observation.

In general, it is clear that dependability benchmarking is still a developing art. It took decades for performance benchmarking to mature, progressing from measures such as Whetstones, Dhrystones, and Livermore Loops to multiple generations of SPEC benchmarks. Primary technical challenges were in the areas of representativeness of the workloads, scalability of benchmarks to larger systems, and vulnerability of benchmarks to performance optimizations by either system design and tuning or aggressive compiler optimizations. It should, therefore, be reasonable to expect that dependability benchmarking will similarly take many years to reach maturity.

There are some aspects of dependability benchmarking that are likely to prove even more challenging than achieving well-accepted performance benchmarks. First, actual performance is something that can be measured in a reasonably short amount of time (hours or days) in an actual production system. Thus, the accuracy of a performance prediction can be validated fairly quickly with respect to a particular user's everyday experience. On the other hand, measuring achieved dependability requires operating for a long enough period of time to observe how the system responds to real failure. Given a reasonably reliable system, it could take months or years to find out if a dependability benchmarking prediction bears any resemblance to actual experience for a given system.

An additional complication with creating a good dependability benchmark is the public unavailability of comprehensive fault and failure data from which to create a faultload. Addressing this lack will at the very least be analogous to the process by which performance benchmarking gathered representative user programs through the creation of the SPEC (Standard Performance Evaluation Corporation) consortium. However, collecting faultloads is likely to be more challenging because many faults depend heavily on implementation and operating environment (they are not portable in the way source programs are). Furthermore, many of the best sources of such data have justifiable competitive disincentives to make data available on what has gone wrong with their systems.

Current trends in computing system design contribute to making dependability benchmarking increasingly difficult. For instance, the highly dynamic nature of most systems and the constant adaptation to changes in the environment, particularly in large networked

systems composed of heterogeneous nodes that demand online deployment of services, runtime reconfiguration, and upgrading, make the benchmarking of dependability attributes a difficult, but extremely important technical challenge. However, the change from single computing nodes to networked systems of systems has also been a challenge for traditional performance benchmarking. Like performance benchmarks, dependability benchmarks will need to continually evolve to keep pace with the changing nature of the computing world.

We expect that eventually there will be a solid set of accepted, standardized, and validated dependability benchmarks. As with performance benchmarking, success will be a journey of continually improving benchmarks to keep up with technological changes, but the rewards of such a journey will be plentiful.

The idea of preparing this book arose during the completion of DBench, a European project on dependability benchmarking, partially supported by the European Commission during 2000–2004 [DBench], and has matured and evolved within the Special Interest Group on Dependability Benchmarking [SIGDeB], founded in 1999 by the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance.

DBench developed a framework for defining dependability benchmarks for computer systems, with emphasis on off-the-shelf components (commercial or not) and on systems based on off-the-shelf components, via experimentation and modeling [Kanoun et al. 2001]. To exemplify how the benchmarking issues can actually be handled in different application domains, a set of benchmarks and their associated implementations has been developed. They concern general-purpose operating systems, embedded systems (automotive and space applications), and transactional systems. These benchmarks are presented in this book. They address specifically the end-user and the system integrator points of view. They share the following common characteristics:

- The benchmark performer is the system purchaser or a third party who has no in-depth knowledge about the benchmark target and who is aiming at obtaining valuable information about the target system dependability.
- The primary users of the benchmark results are the end users of the benchmark target or the *integrators* of the system including the benchmark target.

On the other hand, the work of the Special Interest Group on Dependability Benchmarking started with an open-ended mission of exploration [Koopman and Madeira 1999], but evolved to consider a system vendor point of view [Wilson et al. 2002]. The benchmark performer, who is the system developer, has access to detailed information on the system that is not available for system purchasers. The work has identified a set of standardized classes for characterizing the dependability of computer systems. The classification seeks to enable comparison of different computer systems in the dimensions of availability, data integrity, disaster recovery, and security. Different sets of criteria have been proposed for computer systems that are used for different application types (e.g., transaction processing and process control).

Pioneer work on dependability benchmarking is published in [Tsai et al. 1996] for fault tolerant systems, in [Mukherjee and Siewiorek 1997] for software systems, and in [Brown and Patterson 2000] for Software RAID systems. The first open workshop on the topic was held by the Special Interest Group on Dependability Benchmarking in 2002 [Workshop 2002]. Much of the work on dependability benchmarking was published after 2000, authored primarily by institutions and individuals contributing to this book.

REFERENCES

[Brown and Patterson 2000] A. Brown and D. A. Patterson, "Towards Availability Benchmarks: A Case Study of Software RAID Systems," in *Proceedings 2000 USENIX Annual Technical Conference,* San Diego, CA, USA, USENIX Association, 2000.

[DBench] http://www.laas.fr/DBench.

[Gray 1993] Jim Gray (Editor), *The Benchmark Handbook for Database and Transaction Systems* (2nd Edition), Morgan Kaufmann, 1993.

[Kanoun et al. 2001] K. Kanoun, J. Arlat, D. Costa, M. Dalcin, P. Gil, J.-C. Laprie, H. Madeira, and N. Suri, "DBench—Dependability Benchmarking," in *Supplement of the International Conference on Dependable Systems and Networks,* Göteborg, Sweden, 2001, pp. D.12–D.15.

[Koopman and Madeira 1999] P. Koopman and H. Madeira, "Dependability Benchmarking & Prediction: A Grand Challenge Technology Problem," in *First International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems,* Phoenix, November 1999.

[Mukherjee and Siewiorek 1997] A. Mukherjee and D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking," *IEEE Transactions of Software Engineering,* vol. 23 no. 6, pp. 366–376, 1997.

[SIGDeB] http://www.laas.fr/~kanoun/ifip_wg_10_4_sigdeb/.

[Tsai et al. 1996] T. K. Tsai, R. K. Iyer, and D. Jewitt, "An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems," in *Proceedings 26th International Symposium on Fault-Tolerant Computing (FTCS-26),* Sendai, Japan, pp. 314–323, IEEE Computer Science Press, Los Alamitos, CA, 1996.

[Wilson et al. 2002] D. Wilson, B. Murphy, and L. Spainhower, "Progress on Defining Standardized Classes for Comparing the Dependability of Computer Systems," in *Workshop on Dependability Benchmarking,* pp. F1–F5, Washington, DC, 2002.

[Workshop 2002] Workshop on Dependability Benchmarking, Supplement Volume of 2002 *International Conference on Dependable Systems and Networks (DSN),* July 2002, pp. F1–F36, IEEE Computer Society Press. Also, papers are available at: http://www.laas.fr/~kanoun/ifip_wg_10_4_sigdeb/external/02-06-25/index.html.

1

THE AUTONOMIC COMPUTING BENCHMARK

Joyce Coleman, Tony Lau, Bhushan Lokhande, Peter Shum,
Robert Wisniewski, and Mary Peterson Yost

1.1. INTRODUCTION TO THE AUTONOMIC COMPUTING BENCHMARK

In 2001, IBM initiated a project to revolutionize the self-managing capability of IT systems [Horn 2001]. The company formed a new business unit to execute this mission. Five years later, IBM has achieved great progress in standards evolution, technology innovation, and product deliveries from IBM and throughout the IT industry. One of the key questions we asked in 2001 was, "How can we measure autonomic capability?" A team was assembled with performance benchmarking experts from several product areas to address this question, and the Autonomic Computing Benchmark project was initiated.

This project resulted in the development of the Autonomic Computing Benchmark, which is one of the first benchmarks designed to measure the system resiliency of an enterprise environment. Just as other industry benchmarks allow standardized comparisons between product offerings from competing vendors, we hope that this benchmark will help in quantifying the self-healing capabilities of systems. We believe that this type of quantification is necessary to enable customers to accurately assess resiliency claims from vendors, and to assist vendors in identifying key areas in which they can improve the resiliency characteristics of their products.

The Autonomic Computing Benchmark uses a fault injection methodology and five

Dependability Benchmarking for Computer Systems. Edited by Karama Kanoun and Lisa Spainhower

categories of faults or disturbances.* The initial system under test is composed of a multitier Java™ 2 Platform—Enterprise Edition (J2EE) environment that includes Web server, application server, message server, and database server components. Two metrics are used to evaluate the system resiliency: a quantitative *throughput index* that represents the impact of the disturbances on quality of service, and a qualitative *maturity index* that represents the level of human interaction needed to detect, analyze, and recover from a disturbance, as defined by the IBM Autonomic Maturity Model [IBM 2001].

In this chapter, we describe our experiences designing and executing the Autonomic Computing Benchmark. In Section 1.2, we discuss the requirements that guided our benchmark design. Section 1.3 gives an overview of the Autonomic Computing Benchmark methodology, disturbances, and metrics. Section 1.4 explains how to interpret the results of the benchmark. In Section 1.5, we explain some of the potential uses of the benchmark. Section 1.6 discusses some challenges that we faced during the design and execution of the benchmark. In Section 1.7 we present our conclusions.

1.2. BENCHMARK REQUIREMENTS

Before we set out to develop the Autonomic Computing Benchmark, we agreed that there are several key characteristics and requirements that would ensure an effective and reliable benchmark.†

1.2.1. Concise Metrics

One of the most important requirements guiding the design of the Autonomic Computing Benchmark was that it should produce a small set of metrics that are easily interpreted. These metrics would be used in many ways, such as to compare the same system over time, or to compare different systems with the same business function.

1.2.2. Diverse Levels of Autonomic Maturity

The benchmark should be applicable to systems at any level of autonomic maturity since it might be many years before autonomic features are widely available. This requirement also means that initial testing and results may rank very low on indexes that are measured against "ideal" behavior that is not currently available or even possible with current industry offerings.

In addition, this wide maturity capacity would allow maximum applicability to a diverse range of applications and business needs. Similar to the need for different availability levels based on the importance of an application, we would also want to differentiate maturity based on the requirements of the enterprise system.

*In this chapter, we use the words "fault" and "disturbance" interchangeably. The word "fault" often implies an invalid operation and is the common term used in the dependability literature. The word "disturbance" conveys a broader meaning that covers invalid operations, intrusions, interruptions, and events that could alter the state of the system under test. For example, a load surge of ten times the number of users accessing a system is not an invalid operation but could cause a detrimental impact on the system. However, we continue to use the terms "fault injection" and "performance under fault" because they are commonly used and because they sound better than "disturbance injection" and "performance under disturbance."

The views expressed in this chapter are those of the authors and do not necessarily represent the views of IBM Corporation.

†Sections 1.2.1 through 1.2.4 are adapted from [Lightstone et al. 2003].

1.2.3. Benchmark Suites

Autonomic computing is intended to address a large range of business scenarios and technology elements. The broad implication is that a suite of benchmarks would be needed to cover scenarios such as Web-based e-commerce, data warehousing and decision support, and e-mail. Thus, we define a reference framework that defines phases and metrics for all Autonomic Computing Benchmarks to be included in the suite, based on the notion of a reference workload. A reference workload accommodates the particular business requirements in a scenario, such as the number of transactions processed by a business-to-business (B2B) application, or the number of e-mails processed by a messaging server that match a business scenario. By adding reference workloads to the benchmark specification, the framework is extended to a large number of business classes over well-defined and accepted workloads and schemas.

In addition to the diverse workloads, unique environments also create the need for unique disturbances and administrative interactions. The reference framework should provide the capacity to easily and effectively interact with and fail new components within the system under test.

1.2.4. Low Cost

Benchmarks are very costly to develop. A particular concern was that the Autonomic Computing Benchmark would not be run if the costs were too high. Thus, it is essential to leverage the resources and skills used for existing performance benchmarks in the construction of autonomic computing benchmarks.

1.2.5. Administrative Interactions

Since the very essence of the testing in this scenario is interacting with and many times disabling components in the system under test, it was necessary for the reference framework to provide the capability to restore the system to its previous state and prepare it for further testing.

Measuring the effects of these disturbances may require hours of observation and timing within the system, and, therefore, the ability to fully automate the construction, preparation, and recovery of the system would be required in order to make the toolkit usable and economical. This could be accomplished with a “write once” interaction that could be registered with the framework to perform tasks such as collecting and restoring log information or restarting servers and processes.

1.2.6. The Autonomic Computing Benchmark

The Autonomic Computing Benchmark was designed to meet the five requirements described in Sections 1.2.1 through 1.2.5:

1. The benchmark offers a qualitative, comparable, and concise measurement of autonomic capability that uses two metrics: a quantitative *throughput index* that represents the impact of the disturbances on quality of service, and a qualitative *maturity index* that represents the level of human interaction needed to detect, analyze, and recover from a disturbance, as defined by the IBM Autonomic Maturity Model [IBM 2001].
2. The benchmark is capable of measuring a wide variety of autonomic levels because

of the granularity of the Autonomic Maturity Model. As expected, we have seen low maturity scores in the initial runs because the maturity model leaves room for vast improvement in autonomic features and products over what is currently possible.

3. Because the Autonomic Computing Benchmark is designed to wrap around a reference workload, it is applicable to a number of scenarios. For our initial reference implementation, we selected the SPECjAppServer2004 Performance Benchmark, a popular J2EE performance benchmark from the SPEC organization [SPECj 2004]. Since then, we have extended the Autonomic Computing Benchmark to other workloads and benchmarks.
5. Our use of existing performance benchmarks and workloads makes the Autonomic Computing Benchmark inexpensive to set up and run. The infrastructure required to run the benchmark is simple and lightweight.
6. The Autonomic Computing Benchmark is designed with the capability to automate virtually all administrative tasks in the system under test. Log records can be automatically stored so that, after a run has ended, a benchmark operator can verify that a disturbance ran correctly and investigate how the system under test responded to the disturbance.

1.3. OVERVIEW OF THE AUTONOMIC COMPUTING BENCHMARK

1.3.1. Methodology

The Autonomic Computing Benchmark approaches the task of measurement by using an existing performance benchmark workload, injecting disturbance as the workload is executing, and measuring the performance under fault as compared to a stable environment.

The benchmark methodology consists of three phases, as outlined in Figure 1.1. These are the baseline phase, the test phase, and the check phase. Note that prior to running a baseline phase or test phase, the workload must be allowed to ramp up to steady state, in which the workload runs at a consistent level of performance.

The baseline phase determines the operational characteristics of the system in the absence of the injected perturbations. The running of this phase must comply with all requirements defined by the performance workload.

The test phase determines the operational characteristics of the system when the workload is run in the presence of the disturbances. This phase uses the same setup and configuration as the baseline phase. The test phase is divided into a number of consecutive *fault injection slots.* These fault injection slots are run one after another in a specified sequence.

The check phase ensures that the reaction of the system to the disturbances did not af-

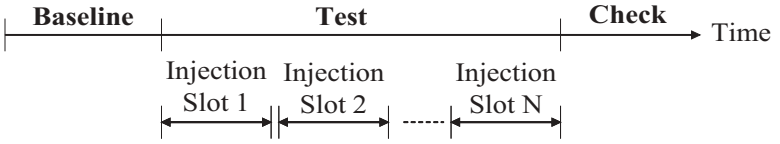

Figure 1.1. Autonomic Computing Benchmark phases.

fect the integrity of the system. During this phase, a check is made to ensure that the system is in a consistent state.*

During each injection slot, the benchmark driver initiates the injection of a disturbance into the system under test (SUT). Ideally, the SUT detects the problem and responds to it. This response can consist of either fixing the problem or bypassing the problem by transferring work to a standby machine without resolving the original problem. If the SUT is not capable of detecting and then either fixing or bypassing the problem automatically, the benchmark driver waits an appropriate interval of time, to simulate the time it takes for human operator intervention, and initiates an appropriate human-simulated operation to recover from the problem.

As Figure 1.2 demonstrates, each injection slot consists of five subintervals:

1. The *injection interval* is the predefined time that the system is allowed to run at steady state before a particular disturbance is injected into the SUT. The benchmark driver waits for the predefined injection interval before injecting the disturbance. The purpose of the injection interval is to demonstrate that the system is functioning correctly before any disturbance is injected.
2. The *detection interval* is the time from when a disturbance is injected to the time when a disturbance is detected. For a SUT that is not capable of detecting a disturbance automatically, the driver will be configured to wait for a predefined detection interval before initiating a recovery action. This is to simulate the time it takes for the human operator to detect a disturbance.
3. The *recovery-initiation interval* is the time from when a disturbance is detected to the time when a recovery action begins. For an SUT that is not capable of detecting the disturbance or initiating a recovery action automatically, the driver will be configured to wait for a predefined recovery-initiation interval before initiating the recovery action. This is to simulate the time it takes for a human operator to initiate recovery.
4. The *recovery interval* is the time that it takes the system to perform recovery. Because the system is allowed to recover during every fault injection slot, all disturbances are considered independently of one another, rather than cumulatively.

*For example, in industry benchmarks such as TPC-C and SPECjAppServer2004, the specifications include requirements to verify that the transaction summary in the workload driver matches the row counts of various tables in the database under test.

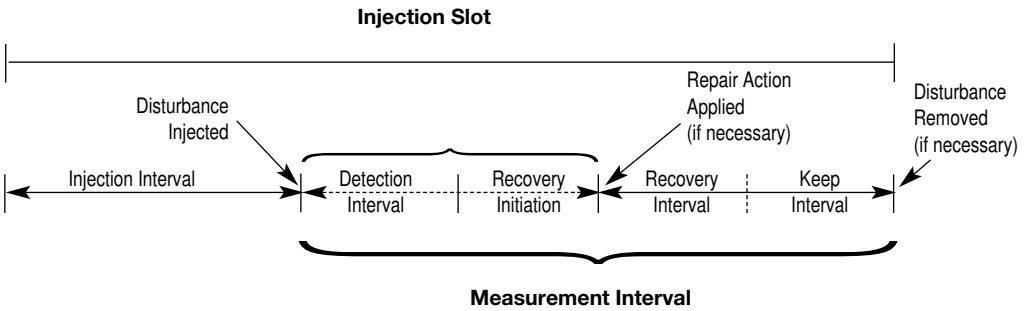

Figure 1.2. Injection slot subintervals.

In other words, no more than one disturbance affects the system at any given time.

5. The *keep interval* is the time to ramp up again and run at steady state after the recovery. This is the time remaining in a measurement interval. It is valid to run the steady state at a different throughput from that used in the injection time period.

It is important to note two things. First, the breakdown of the slot interval into subintervals is for expository purposes only. During a benchmark run, the benchmark driver only distinguishes the boundaries of these subintervals when the system under test requires simulated human intervention. Second, only the operations processed during the last four of the five intervals are part of the measurement interval and are, therefore, counted when calculating the throughput for the run.

1.3.1.1. Practical Considerations Relating to the Methodology. In our implementation of the Autonomic Computing Benchmark, we made one modification to the methodology described in Section 1.3.1. Our initial design goal was to run all of the fault injection slots one after another in a specified sequence. In our experience, however, we found that running multiple disturbances in sequence had the undesirable effect of requiring that the database tier recover for all prior transactions from previous injection slots if it had to perform recovery during the current slot. Therefore, we prefer to run injection slots in isolation, with a check phase following each injection slot.

Regarding the interval lengths that we commonly use, we find that a 50-minute phase interval provides a balance between efficiency and the need to allow a system under test sufficient time to detect and recover from the injected disturbances. Therefore, for a baseline run we allow the system to warm up for 5 minutes, and then use a 50-minute baseline phase. For a test run, we allow the system to warm up for 5 minutes, and then use a 50-minute test phase, which is broken up into 10 minutes for the injection interval, 20 minutes for the combined detection interval and recovery-initiation interval, and 20 minutes for recovery interval and keep interval.

1.3.1.2. Comparison to the DBench-OLTP Methodology. The methodology for our benchmark is in many ways similar to that of DBench-OLTP, a dependability benchmark for OLTP application environments that is one of several benchmark prototypes being developed within the DBench community [DBench 2003]; see also Chapter 5 of this book. Both the Autonomic Computing Benchmark and DBench-OLTP are based on the idea of injecting faults while a workload is executing, and measuring the performance under fault as compared to a stable environment. However, as described by Brown and Shum [2005], there are several important differences between the Autonomic Computing Benchmark and DBench:

- The reference workload for DBench-OLTP is TPC-C. The Autonomic Computing Benchmark, in contrast, can be extended to work with any workload and workload driver, as long as the output is in the form of throughput over time (for example, a count of successful operations every 10 seconds). The Autonomic Computing Benchmark has already been used with SPECjAppServer2004 and with Trade 6, a J2EE workload, using both the IBM WebSphere® Studio Workload Simulator and Rational® Performance Tester to drive the Trade 6 workload.
- DBench-OLTP has a single-component focus, such as a database management server. The Autonomic Computing Benchmark is used to test a multicomponent envi-

ronment, such as a J2EE environment, which allows it to more realistically simulate a customer environment.

- Both DBench-OLTP and the Autonomic Computing Benchmark quantify and measure the performance impact of a fault or disturbance, but the Autonomic Computing Benchmark also quantifies and measures the level of human interaction needed to detect, analyze, and recover from a disturbance.
- DBench-OLTP uses a number of quantitative metrics, including the number of transactions executed per minute in the presence of faults, the price per transaction in the presence of faults, the number of data errors, and two measures of availability. The Autonomic Computing Benchmark uses a single quantitative metric based on throughput.

1.3.2. Disturbance

The disturbances in our execution runs were developed based on customer surveys and internal problem management reports. This list is not intended to be comprehensive, and we intend to refresh it based on customer feedback and experience. Also, we did not orient our work to security issues. System security is a specific discipline that should be addressed fully by experts in this area.

In the next sections, we present the five disturbance categories and describe the disturbances in detail.

1.3.2.1. Unexpected Shutdown. Disturbances in this category simulate the unexpected shutdown of an operating system, one or more application processes, or the network link between components in the SUT (Table 1.1).

TABLE 1.1. Unexpected shutdown faults

Fault name	Description
Abrupt OS shutdown for DBMS, application, HTTP, and messaging servers	This disturbance scenario represents the shutdown of the server operating system. It is intended to simulate the situation in which an operator accidentally issues an operating system shutdown command either remotely or at the console. All the processes on the server are stopped and the operating system is halted gracefully. This is different from the well-known blue screen situation on the Windows® platform (which is considered a software bug), a power failure (which is tied to the hardware), or accidentally shutting down by using the power switch.
Abrupt process shutdown for DBMS, application, HTTP, and messaging servers	This disturbance scenario represents the shutdown of one or more processes supplying the component of the SUT. It is intended to simulate the situation in which an operator accidentally issues an operating system command to end the processes. This is different from issuing a command to the processes to inform them of the need to terminate. The only alert provided to the processes that "the end is near" is that supplied by the operating system to all processes that are to be ended (i.e., signal 9 in Linux).
Network shutdown for DBMS, application, HTTP, and messaging servers	This disturbance scenario represents the shutdown of the network link between critical components of the SUT. It is intended to simulate the situation in which the network becomes unavailable because of a pulled cable, faulty switch, or operating-system-level loss of network control.

1.3.2.2. Resource Contention. Disturbances in this category simulate the case in which resources on a machine in the SUT are exhausted because of an unexpected process, user action, or application error (Table 1.2).

1.3.2.3. Loss of Data. Disturbances in this category simulate a scenario in which business-critical data is lost (Table 1.3).

1.3.2.4. Load Resolution. Disturbances in this category simulate a sudden increase in the workload on the system (Table 1.4).

1.3.2.5. Detection of Restart Failure. Disturbances in this category simulate a situation in which an application or machine is corrupted and cannot be restarted (Table 1.5).

1.3.3. Metrics

Two metrics are used to capture the effect of a disturbance on the SUT: throughput index and maturity index.

1.3.3.1. Throughput Index. The throughput index is a quantitative measure of the quality of service under fault. It describes the relationship between the throughput when the system is infected with a disturbance and the throughput when it is not infected:

$$\text{Throughput index}_i = P_i / P_{\text{base}}$$

where P_i = number of transactions completed without error during fault injection interval i and P_{base} = number of transactions completed without error during baseline interval (no disturbance).

The overall throughput index is calculated by taking the average of the throughput indexes for each individual injection slot. It is a value from zero to one.

It is important to emphasize that the throughput index should not be considered as an availability score. Availability scores are typically well above 99% because they include the concept of mean time to failure. In other words, they are estimated over a long period of time, when the failure of any component in a system is highly unlikely. When the Autonomic Computing Benchmark is run, in contrast, we ensure that a component fails, and measure the throughput over a very short period of time immediately after this planned failure.

1.3.3.2. Maturity Index. The maturity index is a qualitative measure of the degree of autonomic capability. It is calculated based on questions answered by the test operator. The questions relate to the human involvement in the detection, analysis, and recovery of the disturbance (Table 1.6). The scoring system is derived from the IBM Autonomic Computing Maturity Model [IBM 2001]. For each disturbance, the evaluation is based on how the problem is detected, analyzed, and resolved according to a set of questions, using the following scale:

- A is awarded 0 points (basic)
- B0 is awarded 0.5 points (basic/managed)

TABLE 1.2. Faults due to resource exhaustion

CPU hog on DBMS, application, HTTP, and messaging servers	This disturbance scenario represents the case in which the CPU resource on the system is exhausted. It is intended to simulate the situation in which a certain process in the machine stops being a good citizen and takes over all the CPU cycles. All the CPUs on the system are driven to 100% utilization by the hog process.
Memory hog on DBMS, application, HTTP, and messaging servers	This disturbance scenario represents the case in which all the physical memory on the system is exhausted. It is intended to simulate the situation in which a certain process in the machine stops being a good citizen and takes over all the physical memory. All the free physical memory of the system is taken up by the hog process. This disturbance is complicated by the virtual memory system, so the current implementation is to request all physical memory and randomly access within this memory to simulate page requests.
I/O hog on DBMS server	This disturbance scenario represents the case in which the disk bandwidth of the physical drive containing the business data is saturated. It is intended to simulate the situation in which a certain process in the machine stops being a good citizen and creates unplanned heavy disk I/O activities. The disk actuator is busy servicing read or write requests all the time. This should not be confused with the case in which the bandwidth of the I/O bus is saturated.
DBMS runaway query	This disturbance scenario represents the case in which the DBMS is servicing a runaway query. It is intended to simulate the situation in which a long-running, resource-intensive query is accidentally kicked off during operation hours. It should not be confused with a batch of smaller queries being executed.
Messaging server poison message flood	This disturbance scenario represents the case in which the message queue is flooded with many poison messages. A poison message is a message that the receiving application is unable to process, possibly because of an unexpected message format. It is intended to simulate the situation in which the operator configures a wrong queue destination. A large number of poison messages are sent to the message queue. This should not be confused with the case in which the application is causing a queue overflow.
DBMS and messaging server storage exhaustion	This disturbance scenario represents the case in which the system runs out of disk space. It is intended to simulate the situation in which a certain process in the machine stops being a good citizen and abuses the disk quota. All the disk space of the drives containing the business data is taken up by the hog process.
Network hog on HTTP, application, DBMS, and messaging servers	This disturbance scenario represents the case in which the network link between two systems in the SUT is saturated with network traffic. It is intended to simulate the situation in which a certain process in the machine stops being a good citizen and transfers excessive data on a critical network link.
Deadlock on DBMS server	This disturbance scenario represents the case in which a deadlock involving one or more applications leaves a significant number of resources (rows or tables) in the DBMS locked, making them inaccessible to all applications. Any queries on the DBMS that require these locked resources will not complete successfully.
Memory leak in a user application	This disturbance scenario represents the case in which a user application causes a memory leak that exhausts all available memory on the system. It is intended to simulate the case in which a poorly written application is deployed on an application server.

TABLE 1.3. Loss of data

DBMS loss of data	This disturbance scenario represents the loss of a database table. It is intended to simulate the case in which an operator with a connection to the database issues a simple DROP TABLE command accidentally. The table definition and all the data in the table are lost.
DBMS loss of file	This disturbance scenario represents the loss of a database file that contains critical business data. It is intended to simulate the situation in which an operator accidentally issues an operating system command to delete one or more database files that contain data for a particular table. The DBMS can no longer address the file from the file system. This is different from an operating system file-handle loss, which is considered a bug in the operating system
DBMS and messaging loss of disk	This disturbance scenario represents the loss of a physical hard drive that contains the business data. It is intended to simulate the case in which a hard drive is damaged such that the disk controller marks the targeted hard drive as offline.

TABLE 1.4. Load resolution

Moderate load handling and resolution	This disturbance scenario represents the case in which the load on the SUT increases moderately (generally about two times the previous load). It is intended to simulate the situation in which a heavy load is introduced because of a peak season or marketing campaign. The optimal result for this disturbance is to handle the entirety of the new load with the same response time and results as were seen with the previous load. This can be validly accomplished by overplanning the infrastructure, but a more attractive solution is to share the extra infrastructure with a separate system that is not seeing the increased load.
Significantly increased load handling and resolution	This disturbance scenario represents the case in which the load on the SUT increases drastically (generally about 10 times the previous load). It is intended to simulate the situation in which a significantly heavy load is introduced because of a catastrophic event or failure of the primary system. The optimal result for this disturbance is to handle the same amount of business as before without being overwhelmed by the extreme increase in requests. Technologies that illustrate this characteristic are flow control and quality of service monitors.

TABLE 1.5. Restart failure

OS restart failure of DBMS, application, HTTP, and messaging servers	This disturbance scenario represents the case in which the operating system has been damaged and does not restart. It is intended to simulate the case in which a key file or data that is required during the boot process is lost. When the operating system is rebooted, it fails at the point where the key file cannot be loaded.
Process restart failure of DBMS, application, HTTP, and messaging servers	This disturbance scenario represents the case in which the software component fails to restart. It is intended to simulate the case in which a key file or data that is required during the startup process is lost. When the software program is restarted, it fails at the point where the key file or data cannot be loaded.