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-

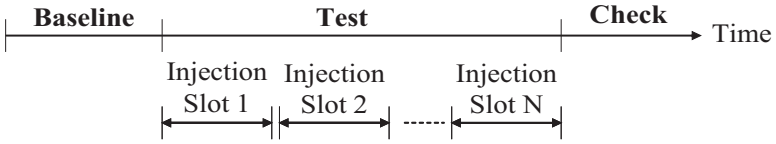

Figure 1.1. Autonomic Computing Benchmark phases.

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.

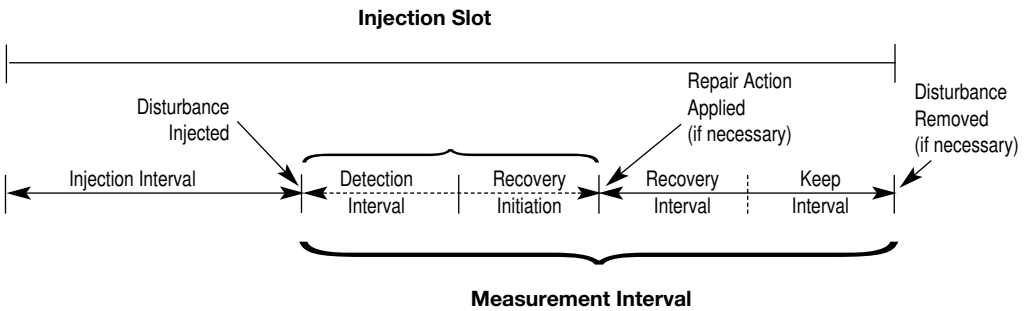

Figure 1.2. Injection slot subintervals.

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.

TABLE 1.6. Maturity index

Autonomic level	Description
Basic	Rely on reports, product, and manual actions to manage IT components
Managed	Management software in place to provide facilitation and automation of IT tasks
Predictive	Individual components and systems management tools able to analyze changes and recommend actions
Adaptive	IT components collectively able to monitor, analyze, and take action with minimal human intervention
Autonomic	IT components collectively and automatically managed by business rules and policies

- B is awarded 1 point (managed)
- C is awarded 2 points (predictive)
- D is awarded 4 points (adaptive)
- E is awarded 8 points (autonomic)

Note that this scale is nonlinear in order to take into account the difficulty of implementing a system that meets the higher autonomic levels.

The questions used are as follows.

How is the disturbance detected?

A. The help desk calls the operators to tell them about a rash of complaints.
B0. The operators detect the problem themselves by monitoring multiple data sources.
B. The operators detect the problem themselves by monitoring a single data source.
C. The autonomic manager notifies the operator of a possible problem.
D. The autonomic manager detects the problem without human involvement.

How is the disturbance analyzed?

A. The operator collects and analyzes multiple sources of system-generated data.
B. The operator analyzes data from a single management tool.
C. The system monitors and correlates data that leads to recommended recovery actions.
D. The system monitors and correlates data that allows actions to be taken without human involvement.
E. The system monitors and correlates data based on business rules and policies that allow actions to be taken without human involvement.

What is the action taken?

A. The operator performs the required procedures and issues the commands on each affected resource individually.
B. The operator performs the required procedures and issues the commands on a centralized management console.

C. The operator approves and initiates the recovery actions.
D. The autonomic system initiates the recovery actions. No human action is needed.

The overall maturity index is the average score of all injection slots normalized to the highest autonomic level possible. It is a value from zero to one. A value of zero indicates that the autonomic capabilities of the system are basic (manually managed by reports, product manuals, and manual actions). A value of one indicates that the system is autonomic (automatically manages itself to achieve business objectives).

1.4. SAMPLE RESULTS

In this section, we present sample results that were obtained during a run of the Autonomic Computing Benchmark using Trade 6, a J2EE workload, as the reference workload.

1.4.1. Baseline Run

Figure 1.3 shows a sample baseline result. The throughput calculated during the baseline phase is used to calculate the throughput index for each disturbance run during the test phase.

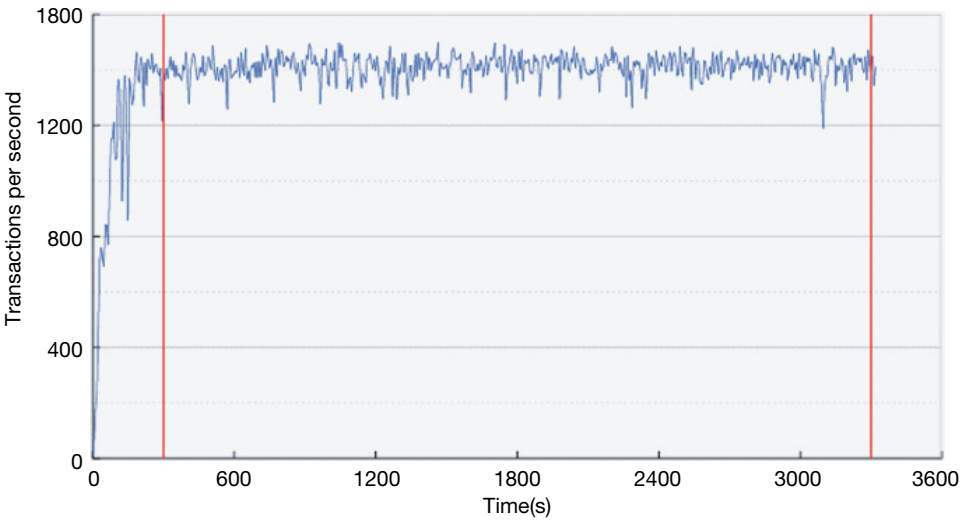

Figure 1.3. Sample baseline result.

1.4.2. Sample Disturbance #1

Figure 1.4 illustrates results that were obtained from a sample test phase run of the Autonomic Computing Benchmark. At the time indicated by the second vertical line, a disturbance that targets one component is injected into the SUT. In this example, the system is not able to remove the effect of the disturbance, and throughput remains close to zero until the third vertical line, when the disturbance is removed from the system (for example, by killing a resource-hog process or by restarting an application that was abruptly shut

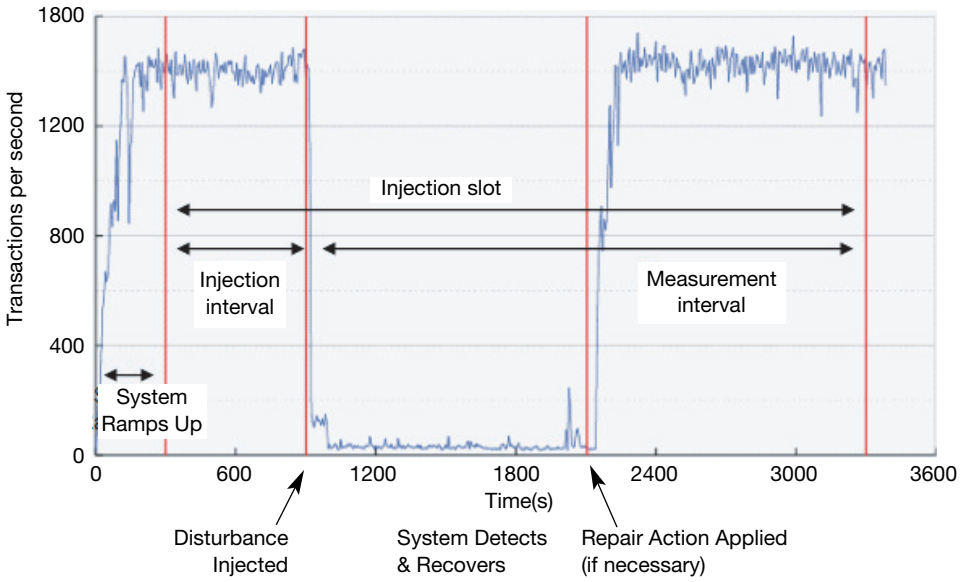

Figure 1.4. Sample disturbance result #1.

down). However, the good news for this SUT is that once the disturbance is removed from the system, throughput quickly returns to normal levels. Note that this might not always be the case. For example, in some cases a disturbance might cause a component in the system so many problems that it is unable to continue processing work even when the disturbance has been removed.

The throughput is calculated by counting the number of successful operations that take place during the measurement interval, and then dividing by the duration of the injection slot. In this example, assume that the throughput is calculated at an average of 700 operations per second. To calculate the throughput index, this value is divided by the value for an equivalent baseline phase. From Figure 1.3, we see that the average throughput during the baseline phase was approximately 1400 operations per second. Therefore, we obtain a throughput index of 0.50. Note that this throughput index is highly dependent on the timings chosen for the benchmark run. If we had kept the same injection interval but shortened the detection initiation interval (i.e., applied the repair action earlier), we would have obtained a higher throughput index. Benchmark runs are, therefore, only comparable when the same intervals are used in each run.

The operator running the benchmark then calculates the maturity index by considering the qualitative response of the system to the disturbance. In this example, it is clear that the SUT is not initiating a recovery action of any kind, but it might still obtain a non-zero maturity index if it detects the problem and perhaps provides a recommendation to the user about a possible recovery action that the user could manually execute.

1.4.3. Sample Disturbance #2

Figure 1.5 demonstrates a repeated execution of the same disturbance, but this time a monitoring product is used to detect the disturbance and initiate a recovery action. It might do this in one of several ways. It might terminate a rogue process that was hogging

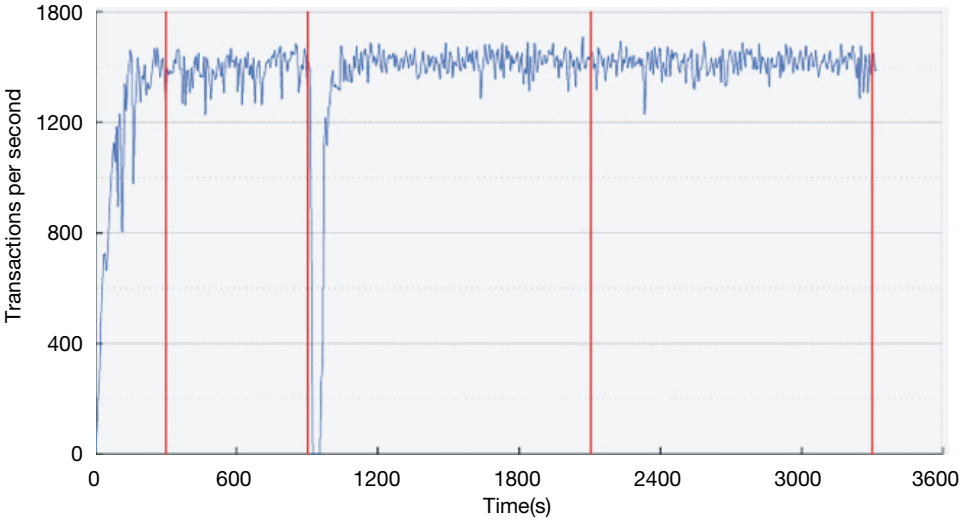

Figure 1.5. Sample disturbance result #2.

a resource in the system (repair), or it might shift work to a standby machine without resolving the original problem (bypass).

In this example, we see that the throughput index will be somewhere on the order of 0.95, since the throughput was zero only briefly. And we will also see a higher maturity score than in the previous example, since the SUT is clearly detecting the disturbance and initiating a recovery action of some kind.

1.4.4. Comparison of Results from Benchmark Runs

One of the possible applications of the Autonomic Computing Benchmark is to assess the value added by an autonomic product to a system under test. In the following example, we compare two executions of the benchmark, using Trade 6 as the reference workload. In the first execution, a set of disturbances was run in a basic SUT. The SUT was then enhanced by adding an autonomic product and the benchmark was executed a second time. The results obtained are shown in Figure 1.6.

From the figure, it can be seen that the enhancement added some value to the SUT. In terms of the throughput index, it had a small but measurable impact, and although the increase in the maturity index might seem small, remember that the scale is nonlinear. In this case, the enhancement raised the SUT from a managed level to a predictive level, which represents a significant improvement in the ability of the SUT to manage itself. A managed system only provides some tools that allow a user to monitor the system and initiate actions, whereas a predictive system is one that can analyze incoming data and make recommendations to the user about possible actions to take.

An analysis of the above results could be that the enhancement increased the maturity level significantly by detecting and analyzing the disturbances, but did not have a significant impact on throughput because it could not initiate automatic recovery actions. In other words, this particular enhancement can simplify the work of human operators monitoring the system, but is not able on its own to replace those human operators.

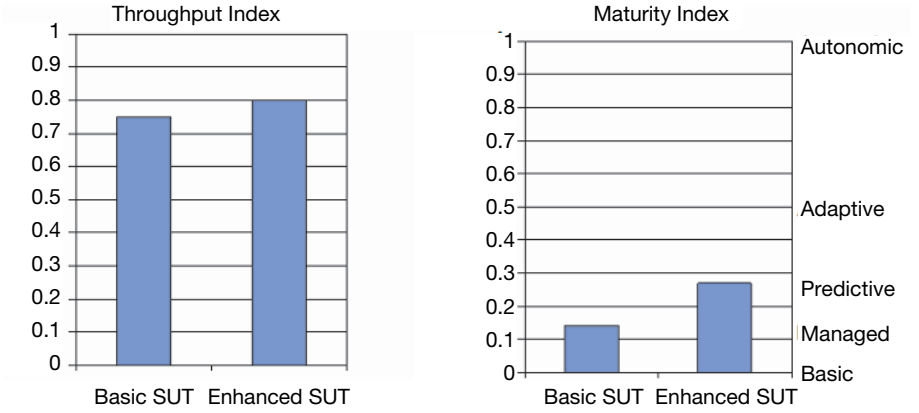

Figure 1.6. Comparison of basic SUT and enhanced SUT.

1.5. APPLICATIONS OF THE AUTONOMIC COMPUTING BENCHMARK

We believe that the Autonomic Computing Benchmark has a number of potential uses for a variety of different types of users.

IT infrastructure specialists could use the benchmark to:

- Test the resiliency of new applications (whether developed in-house or purchased)
- Automate testing in their environment to determine areas of weakness
- Set and evaluate resiliency goals for products

Vendors could use the benchmark to:

- Measure version-to-version improvements in their products
- Focus their development efforts on designing state-of-the-art autonomic features that target the areas of greatest need
- Generate marketing metrics and statistics that demonstrate the resiliency of their products
- Publish results in order to highlight strengths as compared to other vendors

System integrators could

- Measure cross-product resiliency
- Compare a system with and without a component (for example, a monitoring product or failover support) in order to quantitatively assess the value added by the feature

1.6. CHALLENGES AND PITFALLS

Although the benchmark provides an excellent way of quantifying the resiliency of a system, it also presents a number of challenges.*

*Sections 1.6.1, 1.6.3, and the introduction to 1.6.4 are borrowed from [Brown et al. 2004]. Sections 1.6.4.1 through Section 1.6.4.3 are based on an unpublished draft article that was abbreviated and converted into the poster paper [Brown and Redlin 2005].

1.6.1. Design of Disturbances

There are three keys challenges in designing disturbances. The first is to select the set of disturbances to implement so that they are meaningful and relevant to customer scenarios. The second is to ensure that the benchmark remains reproducible despite injecting the disturbance. We must ensure that individual disturbances can be injected reproducibly, and must coordinate injected disturbances with the applied workload. The third challenge concerns the representativeness of the injected disturbances. Unlike a performance workload, which can be simulated in isolation, environmental disturbances might require a large-scale simulation infrastructure. Significant resources may be needed to successfully inject these disturbances, unless tricks can be found to simulate their effects with fewer resources.

We have responded to these challenges by basing our set of disturbances on customer surveys and internal problem management reports, and by having subject-matter experts from each of the components in the SUT design and implement the disturbances. Through several iterations of the benchmark on different SUTs, we have found that well-designed disturbances can be injected reproducibly. In some cases, we have faced difficulties in injecting large-scale disturbances, and we continue to search for methods of simulating disturbances without necessitating significant resources.

1.6.2. Comparing Results

The flexibility of the Autonomic Computing Benchmark raises the question of what types of comparisons between results are meaningful. In order to compare the resiliency of two SUTs, the reference workload, disturbances, and timing parameters used in the runs must clearly be identical. But what if, for example, the user wishes to compare how the products from vendors A and B react to an abrupt shutdown? It will be necessary to write two product-specific implementations of the abrupt shutdown disturbance. Judgment is required to ensure that the two implementations are similar enough in design that the comparison is fair to both products.

Another consideration is how systems of different scale can be compared. A large, high-throughput system will take longer to crash, and will require a longer interval in which to recover after the crash. A possible solution to this problem is to create classes of SUTs based on throughput or on different scaling factors in the workload, such as number of users and number of products, and to compare only results from within a single class.

A third factor complicating the comparison of results is the absence of a cost component in the metric, which makes it possible for an operator to artificially inflate a resiliency score by overprovisioning hardware and software components in the SUT. Currently, we require only that a benchmark result include a list of hardware and software components used to achieve that result. One way to minimize overprovisioning as well as to ensure that the system under test is reasonably stressed is to require a minimal level of utilization from the key components.

1.6.3. Handling Partially Autonomic Systems

Partially autonomic systems include some autonomic capabilities, but still require some human administrative involvement to adapt fully. For example, a system might diagnose a problem and suggest a course of action, but wait for an administrator's approval before completing the self-healing process. An autonomic computing benchmark should provide

useful metrics for such systems so that we can quantify the steps toward a fully autonomic system. But the benchmark cannot easily simulate a representative, reproducible human administrator to complete the SUT's autonomic loop. Human involvement in the benchmark process itself may need to be considered, in which case the benchmark scope expands to include aspects of human-user studies, with statistical techniques used to provide reproducibility. An alternative approach is to break the benchmark into separate phases such that human intervention is only required between phases. Each phase would then be scored individually, with a scorecard-based penalty applied according to the amount of interphase human support needed. This phase-based approach also may help with benchmarking of systems that crash or fail in response to injected changes, because failure in one phase can be treated independently from behavior in other phases.

1.6.4. Metrics and Scoring

Autonomic benchmarks must quantitatively capture four dimensions of a system's autonomic response:

- The level of the response (how much human administrative support is still needed)
- The quality of the response (whether it accomplishes the necessary adaptation)
- The impact of the response on the system's users
- The cost of any extra resources needed to support the autonomic response

The quality and impact dimensions can be quantified relatively easily by measuring end-user-visible performance, integrity, and availability both during and after the system's autonomic response. The level dimension is harder to quantify, as it requires an assessment of human involvement with the system.

There are a number of challenges surrounding the issue of metrics and scoring. In the following sections, we expand on several metric-related issues.

1.6.4.1. Quantifying the Impact of a Disturbance on an SUT. A key issue when measuring self-healing involves the details of how to quantify the impact of the injected disturbance on the SUT. In a non-self-healing SUT, the impact will be visible in the quality of the SUT's response, motivating use of a quality of service (QoS) metric. But measuring QoS has many subtleties, and the benchmark designer has to make choices. There are many possible QoS metrics—throughput, response time, correctness, and so on. Which should be used? Should the benchmark measure the SUT's response as a pass/fail test against a predefined QoS limit, or should it compute the degree of QoS degradation? Should the QoS evaluation be done against a single, fixed level of applied workload (e.g., a certain number of requests per second chosen to drive the SUT at a predefined fraction of its full capacity), or should the benchmark instead test a full range of workloads in order to expose how the SUT's behavior varies at different utilization levels?

To measure the impact of a disturbance to the SUT's delivered QoS, we chose to measure QoS in terms of throughput, using a fixed applied workload. This kind of throughput measure is often known as "goodput."

We measure throughput relative to a workload generated by a constant number of simulated users each making requests at a fixed maximum rate. Note that each simulated user runs on its own thread (or threads) of execution. When a disturbance causes the SUT to

slow down to the point where it cannot satisfy the maximum rate of incoming requests, some threads will block and the request rate will decrease to match the SUT's capacity. In our experiments, we chose the number of users to generate a throughput that matches the capacity of a system that has one fewer cluster member than the SUT being exercised. This choice prevents the SUT from saturating when some of the disturbances are applied. This choice of a workload allows a distinction to be made between those disturbances that significantly reduce the capacity of the SUT, those that result in a total loss of service, and those that do not impact the ability of the SUT to service requests. Our QoS measure is also able to provide insights into which disturbances take a long time for recovery.

1.6.4.2. Quantifying the Level of Autonomic Maturity. One crucial aspect of autonomic computing is the ability of the SUT to self-heal automatically. Since very few SUTs will have purely autonomic or purely manual healing processes, the benchmark must be able to quantify how close the SUT is to fully automated self-healing. Conversely, it must quantify the degree of involvement of human administrators in the healing process. Figuring out how to quantify the level of autonomic maturity is probably the most challenging of the issues. Solutions range from qualitative, subjective approaches like scoring a questionnaire filled out by the benchmark user, to quantitative but still subjective approaches like assigning numerical scores to each human action required in an SUT's healing response, to quantitative and objective approaches like measuring the configuration complexity of healing tasks [Brown and Hellerstein 2004].

We chose a subjective measure in the form of a questionnaire to identify the activities required for discovering and recovering from the various disturbances. We classified the activities according to a management model representing five levels of maturity, and assigned a nonlinear point scale to the classifications (reflecting the relatively greater impact of higher levels of maturity). The levels in this scale are based on the IBM Autonomic Computing Maturity Model, and were discussed in Section 1.3.3. For each disturbance that the benchmark injects, the operator running the benchmark fills out a list of questions that help to determine into which classification the SUT falls. The results of the questionnaire are used to determine the maturity level of the autonomic computing capabilities for each disturbance (and, hence, the number of points assigned).

However, because of the subjectivity of our questionnaire, we sometimes have difficulty reaching consensus on how particular disturbances should be scored. In some cases, we find that the IBM Autonomic Computing Maturity Model does not provide enough guidance about how a disturbance should be graded. For example, the difference between an adaptive and autonomic analysis is that an autonomic analysis is "based on business rules and policies," but insufficient guidelines are given about what these business rules and policies should look like.

Another observation from our experiment is that the nonlinear scoring of the autonomic maturity level is too idealistic. Although in theory this is intended to distinguish the relatively greater impact and difficulties of achieving the adaptive and autonomic level, in practice we have found that systems do quite well if they achieve the predictive level. The current nonlinear scale does not give enough credit to the predictive level. In the future, we will experiment with replacing the nonlinear scale (e.g., 0, 1, 2, 4, 8) with a linear scale instead (e.g., 0, 2, 4, 6, 8).

1.6.4.3. Quantifying the Quality of a Self-Healing Action. One more dimension to determine the maturity is to quantify the quality of a self-healing action. In some cases, a self-healing system has no observable impact from a disturbance due to it being a

nonsignificant experiment. In other cases, a self-healing system simply tolerates a disturbance with no adverse effects, such as when a runaway SQL query is prevented from hogging the CPU due to either the design of the operating system's scheduler or the workload management facility of the database system. In yet other cases, self-healing can involve a multistage healing process; the resource affected by the disturbance is first bypassed, so that the effects of the disturbance are no longer visible to users of the system. Next, the affected resource is repaired, and then it is finally reintegrated back into the SUT.

As an example, consider a self-healing storage system based on a redundant array of independent disks (RAID). After a disk failure (the disturbance), the RAID array can bypass the problem by servicing requests from the remaining disks. If the healing process ends here, the RAID array continues to operate, but at a lower level of redundancy, making it susceptible to future disturbances. If the healing process continues, the array initiates repair by reconstructing the failed disk's data on a standby spare disk. Finally, it reintegrates the reconstructed disk back into the system, restoring full redundancy.

Unlike the RAID example, not all SUTs will provide the full set of self-healing stages. Some may simply bypass problems, leaving themselves unprotected against future disturbances. Others may jump straight to recovery. And reintegration may or may not be automatic. A self-healing benchmark must decide how it will handle this panoply of SUT behaviors: should it distinguish between the self-healing stages or measure them as a unit? If they are treated as a unit, how will the benchmark fairly compare an SUT that only does bypass with an SUT that does bypass, repair, and reintegration? If they are measured individually, how will the benchmark draw the dividing line between stages?

We resolve this issue by judging that merely bypassing a problem does not constitute a full recovery. In the case of disturbances for which our SUT can bypass the problem, we calculate the maturity index by taking the average of two scores: one based on the quality of the bypass action, and one based on what repair action (if any) is performed on the initially damaged resources. We base this method on the observation that when no repair occurs, the original set of resources is no longer available at the end of the injection slot. Even if the throughput has returned to its predisturbance level, the state of the SUT is inferior to its original state since it would be unable to withstand a subsequent injection of the disturbance under study.

Regarding the case of a disturbance that is non-significant or is tolerated by an SUT, we currently assign a throughput index of 1.00 and a maturity index of "N/A." We assign this score because the disturbance has no visible impact on the SUT, and, therefore, there is no need for the SUT to either detect or recover from the disturbance. However, this also means that systems that are immune to a particular disturbance by design will receive no credit in the maturity score. In the future, we intend to reevaluate our approach to scoring this kind of scenario.

1.7. CONCLUSION

We have demonstrated a working and effective benchmark for measuring the autonomic self-healing capability of a J2EE-based computing system. Our benchmark quantifies the ability of the SUT to adapt to disturbances injected into its environment in an autonomic way. The benchmark has the potential to play a key role in the engineering of autonomic systems, as it provides the quantitative guidance needed to evaluate new and proposed autonomic technologies, to set goals and development targets, and to assess and validate autonomic progress.

In the future, we hope to accomplish three goals. First, we would like to extend our set of disturbances in order to keep them meaningful and current. This will involve soliciting input from customers and probing the marketplace in order to determine new or existing problem scenarios that could potentially cause customers to experience outages. Second, we would like to drive the lessons learned from our benchmark runs into formal requirements for the next generation of autonomic solutions throughout the industry. (For information on IBM's current autonomic efforts, see [IBM 2006].) And third, we hope to promote the standardization of the field of autonomics by encouraging the adoption of the Autonomic Computing Benchmark by teams within and outside of IBM.

We have identified significant opportunities for refinement and improvement, and have laid out a roadmap for the next generation of autonomic solutions. We look forward to seeing increasingly sophisticated benchmarks for autonomic capability being used widely in autonomic engineering efforts.

ACKNOWLEDGMENT

Note that some of the material from Sections 1.2 and Sections 1.6 is adapted directly from earlier articles by members of the Autonomic Computing team, including [Lightstone et al. 2003], [Brown et al. 2004], and [Brown and Redlin 2005]. We have made minor modifications to the text from these articles and in some cases expanded on it. Within the chapter, we indicate which sections were drawn from the earlier articles. We thank the authors of these articles for allowing us to use their words.

REFERENCES

[Brown et al. 2004] A.B. Brown, J. Hellerstein, M. Hogstrom, T. Lau, S. Lightstone, P. Shum, and M.P. Yost, "Benchmarking Autonomic Capabilities: Promises and Pitfalls," in *Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004),* pp. 266–267, New York, May 2004.

[Brown and Hellerstein 2004] A.B. Brown and J. Hellerstein, "An Approach to Benchmarking Configuration Complexity," in *Proceedings of the 11th ACM SIGOPS European Workshop,* Leuven, Belgium, September 2004.

[Brown and Redlin 2005] A.B. Brown and C. Redlin, "Measuring the Effectiveness of Self-Healing Autonomic Systems," in *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC 2005),* pp. 328–329, Seattle, June 2005.

[Brown and Shum 2005] A.B. Brown and P. Shum, "Measuring Resiliency of IT Systems," presentation to the Workshop on Dependability Benchmarking organized by the Special Interest Group on Dependability Benchmarking (SIGDeB), Chicago, November 2005, http://www.laas. fr/~kanoun/Ws_SIGDeB/5-IBM.pdf.

[DBench 2003] "DBench-OLTP: A Dependability Benchmark for OLTP Application Environments—Benchmark Specification," Version 1.1,0, http://www.dbench.org/benchmarks/DBench-OLTP.D1.1.0.pdf, 2003.

[Horn 2001] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," http://www-03.ibm.com/autonomic/pdfs/autonomic_computing.pdf, 2001.

[IBM 2001] IBM Corporation, "Autonomic Computing Concepts," http://www.ibm.com/autonomic/pdfs/AC_Concepts.pdf, 2001.

[IBM 2006] IBM Corporation, "IBM Autonomic Computing Library," http://www.ibm.com/autonomic/library.shtml, 2006.

[Lightstone et al. 2003] S. Lightstone, J. Hellerstein, W. Tetzlaff, P. Janson, E. Lassettre, C. Norton, B. Rajaraman, and L. Spainhower, "Towards Benchmarking Autonomic Computing Maturity," in *IEEE International Conference on Industrial Informatics (INDIN 2003): Proceedings,* pp 51–59, Banff, Alberta, Canada, 2003.

[SPECj 2004] "SPECjAppServer2004 Design Document", Version 1.00, http://www.spec.org/jAppServer2004/docs/DesignDocument.html, 2004.