I-Queue: Smart Queues for Service Management

Mohamed S. Mansour[1], Karsten Schwan[1], and Sameh Abdelaziz[2]

[1] The College of Computing at Georgia Tech, Atlanta GA 30332, USA
{mansour, schwan}@cc.gatech.edu
[2] Worldspan, L.P., Atlanta GA 30339, USA
sameh.abdelaziz@worldspan.com

Abstract. Modern enterprise applications and systems are characterized by complex underlying software structures, constantly evolving feature sets, and frequent changes in the data on which they operate. The dynamic nature of these applications and systems poses substantial challenges to their use and management, suggesting the need for automated solutions. This paper considers a specific set of dynamic changes, large data updates that reflect changes in the current state of the business, where the frequency of such updates can be multiple times per day. The paper then presents techniques and their middleware implementation for automatically managing requests streams directed at server applications subjected to dynamic data updates, the goal being to improve application reliability in face of evolving feature sets and business data. These techniques (1) automatically detect input patterns that lead to performance degradation or failures and then (2) use these detections to trigger application-specific methods that control input patterns to avoid or at least, defer such undesirable phenomena. Lab experiments using actual traces from Worldspan show a 16% decrease in frequency of server restarts when using these techniques, at negligible costs in additional overheads and within delays suitable for the rates of changes experienced by this application.

1 Introduction

The complexity of modern enterprise systems and applications is causing renewed interest in ways to make them more reliable. Platform virtualization [1] and automated resource monitoring and management [2,3] are system-level contributions to this domain. Middleware developers have introduced new functionality like automated configuration management [4], improved operator interfaces like Tivoli's 'dashboards' [5], automated methods for performance understanding and display [6], and new methods for limiting the potential effects of failures [7,8]. Large efforts like IBM's Autonomic Computing and HP's Adaptive Enterprise initiatives are developing ways to automate complex management or configuration tasks, creating new management standards ranging from Common Base Events for representing monitoring information [3] to means for stating application-level policies or component requirements (e.g. WSLA[9]).

This paper addresses service failures in distributed applications. The failure model used is typical for distributed enterprise applications like web services,

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 252–263, 2006.

where 'failures' are not direct or immediate system or application crashes, but cause atypical or unusual application behaviors captured by distributed monitoring techniques [10]. Examples include returns of empty or insufficient responses, partially correct results, performance degradation causing direct or increasingly probable violations of delay guarantees specified by SLAs, and others.

Focusing on enterprise systems with reliable hardware infrastructure but potentially unreliable software, we investigate ways in which they can deal with unusual behaviors and eventually, failures caused by single or sequences of application requests, which we term *poison* request sequences. In earlier work, we identified and found ways to deal with a simple case observed by one of our industry partners, which concerned single requests, termed a 'poison message' that consistently caused unusual system and application responses [8]. In this paper, we tackle the more complex problem of sets or sequences of requests that cause such behaviors, and where such problems may depend on dynamic system conditions, such as which business rules are currently being run or to what current states they are being applied. The specific example studied is a global distribution system (GDS) that does transaction processing for the travel industry. To summarize, our assumption is that even with extensive testing applied to modern enterprise applications, it is difficult, if not impossible to ensure their correct operations under different conditions. This is not only because of the undue costs involved with testing such systems under all possible input sequences and application states, but also because the effects of poison message sequences can expose hidden faults that depend both on the sequence of input messages and on changes in system state or application databases. Examples of the latter include regular business data updates, evolving application databases, and system resources that are subject to dynamic limitations like available virtual memory, communication buffers, etc.

The particular problem considered in this paper is *poison requests* or request sequences arriving at a server system. These sequences lead to corrupted internal states that can result in server crash, erroneous results, degraded performance, or failure to meet SLAs for some or all client requests. To identify such sequences, we monitor each single server, its request sequences and responses, and its resource behavior. Monitoring results are used to dynamically build a library of sequence patterns that cause server failures. These techniques use dynamic pattern matching to detect poison sequences. While failure detection uses general methods, the techniques we use for failure prevention exploit application semantics, similar to what we have done in our earlier work on poison messages and more generally, in our 'Isolation-RMI' implementation of an improved communication infrastructure for Java-based enterprise infrastructures like Websphere or JBOSS [8]. As with solutions used to improve the performance of 3-tier web service infrastructures [11], we simply interpose a request scheduler between clients and server. In contrast to earlier work on load balancing [11], however, the purpose of our scheduler is to detect a potentially harmful request sequence and then change it to prevent the failure from occurring or at least, to delay its occurrence, thereby improving total system uptime. One specific prevention method used in this paper is to shuffle requests or change request order to defer (or eliminate) an imminent server crash. The idea is to dynamically apply different request shuffling methods within some time window, to prevent a failure or to at least, opportunistically defer it, thereby reducing the total time spent on system recovery or reboot.

Our motivation and experimental evaluation are based on a server complex operated by Worldspan, which is a leading GDS and the global leader in Web-based travel e-commerce. Poison message sequences and their performance effects were observed in a major application upgrade undertaken by the company in 2005, after a one man-year development effort for which its typical internal testing processes were used. The failures observed were degraded system performance resulting from certain message sequences, but system dynamics and concurrency made it difficult to reproduce identical conditions in the lab and identify the exact sequence and resource conditions that caused the problem. The current workaround being used is similar to the micro-reboot methods described in [12]. The experimental work described in this paper constitutes a rigorous attempt to deal with problems like these, using requests, business software, and request patterns made available to our group by this industry partner.

A concrete outcome of our research is the *I-Queue* request management architecture and software implementation. I-Queue monitors a stream of incoming Web Service requests, identifies potential poison message sequences, and then proactively manages the incoming message queue to prevent or delay the occurrence of failures caused by such sequences. The I-Queue solution goes beyond addressing the specific server-based problem outlined above, for multiple reasons. First, I-Queue is another element of the more general solution for performance and behavior isolation for distributed enterprise applications described in [8]. The basic idea of that solution is to embed performance monitoring and associated management functionality into key interfaces of modern enterprise middleware: (1) component interfaces, (2) communication substrates like RMI, and (3) middleware-system interfaces. Here, I-Queue is the messaging analogue of our earlier work on I-RMI [8]. Second, I-Queue solutions can be applied to any 3-tier web service infrastructure that actively manages its requests, an example being the popular RUBiS benchmark for which other research has developed request queuing and management solutions to better balance workloads across multiple backend servers. In that context, however, I-Queue's dynamic sequence detection methods would be embedded into specific end servers or into queues targeting certain servers rather than into the general workload balancing queue containing all requests in the system. Otherwise, substantial overheads might result from the need to sort requests by target server ID. Third, I-Queue solutions can be applied to request- or message-based systems, examples of the latter including event-based or publish-subscribe systems [13] or messaging infrastructures [14,15].

2 Motivating Scenario

Figure 1 shows an overview of the major components of Worldspan's distributed enterprise applications and systems. The airlines publish their fares, rules and availability information to clearing warehouses (CW). The CW in turn publishes the updates to several GDSs. The GDS implements several services which for a given travel itinerary searches for the lowest available fare across multiple airlines. It is estimated that the size of fare and pricing database at Worldspan, is currently at 10GB and is expected to increase by approximately 20% over the next few years. Worldspan receives an average of 11.5 million queries per day with contractual agreements to generate a reply within a predetermined amount of time. The high message volume coupled with constantly changing system state creates a real need for monitoring and reliability middleware that can learn the dynamically changing performance characteristics and adapt accordingly.

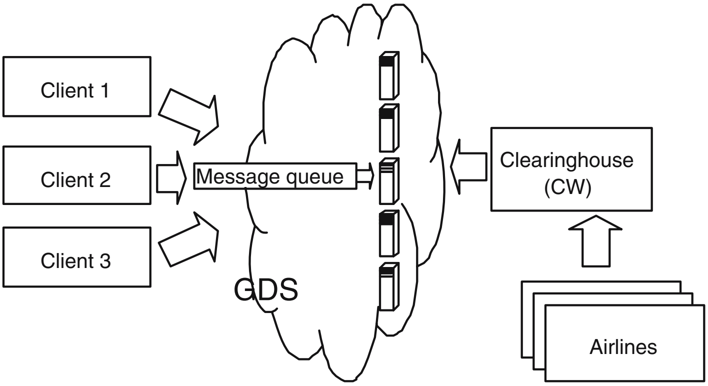

Fig. 1. General overview of message flows in air reservation systems

3 System Architecture

I-Queue uses a simple monitor-analyze-actuate loop similar to those described in previous adaptive and autonomic computing literature [16,17]. Our contribution is adding a higher level analysis module that monitors message traffic and learns the message sequences more likely to cause erratic behavior then apply application specific methods to prevent or reduce the likelihood of such problems.

The monitoring component observes inputs and outputs of the system. The analysis module in our system is the Learning Module (LM), which performs sensitivity analysis by correlating various system performance metrics and input message parameters. The goal of the Learning Module is to establish a set of parameter(s) that can act as good predictors for abnormal behaviors. The output is an internal model that can be used to predict performance behavior for incoming message streams. LM is modular and can use any machine learning algorithm suited for the problem at hand. This paper experiments with algorithms that use Markov Models.

The actuator component is the Queue Management Module (QMM). Using the internal performance model generated by LM, QMM prescans the incoming messages in a short window to see if they are likely to cause performance problems.

If a suspicious sequence is detected, QMM takes an application-specific action to prevent this problem from occurring, or at least, to defer it. The action used in this paper is to re-arrange the buffered messages to another sequence that is not known to cause performance problems, or that is known to cause fewer problems. Figure 2 shows an overview of the system architecture.

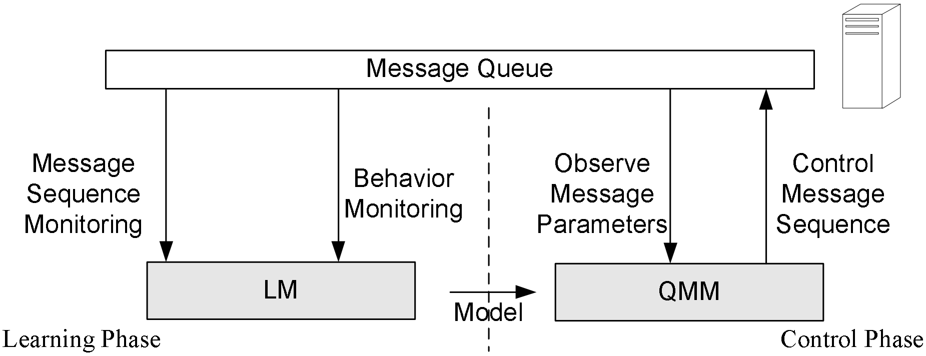

Fig. 2. I-Queue System Architecture

3.1 Internal Design

To demonstrate the value of I-Queue, we used Hidden Markov Models (HMMs) [18] to implement the LM. In our traces, each message is completely independent of other messages, and messages can be processed in any order without changing their semantics. During the learning phase, we construct transition matrices for each observed parameter(e.g., message size, internal message parameters, message inter-arrival time, ...). A transition matrix is a 2D matrix, for each message pair and a specific parameter, where the value of the parameter from the first message indicates the matrix column, and the value of the parameter from the second message indicates the matrix row. Analyzing message pairs leads to a first order model. For an N-order model, we check $N+1$ messages, the concatenated parameter values from messages 1 to N indicate the column and the parameter value from message $N+1$ indicate the row. To reduce matrix size, we use a codebook to convert parameter values to a numeric index. For multi-valued parameters (i.e., list parameters), we use a two level codebook, where the first level encodes each value in the array, then we combine the array values for a message in sorted order and use that for a lookup into the second level codebook. N-dimensional parameters can be dealt with using $N+1$ levels of codebooks.

We currently construct one transition matrix per parameter, but support for combinations of parameters can be added. For each message, we record all parameters as they arrive, and we observe system state after they are processed. If the system ends in a positive state, then we increment the corresponding cells in the transition matrices. If the server crashes or otherwise shows any performance misbehaviors, then we decrement the appropriate cells. During this training period, we also calculate a prediction error rate. This rate gives us an indication of the quality of a parameter as a predictor. It is calculated by counting the number of times the transition matrix for a certain parameter indicates strong likelihood of performance problems that do not actually occur (think of it as a false alarm rate).

An example of a transition matrix is shown in Table 1. For our experiments, we use a second order Markov Model. The rows of the matrix are labeled with the codes from 2 consecutive messages, the columns are labeled with the message that follows in sequence. The cell values give us an indication of server behavior as it executes a particular sequence of messages. A positive value indicates good behavior, e.g., message sequence *AAA* (first row by first column) and the higher the value the better, e.g., *AAA* is more preferable than *AAE* (first row by fifth column). A negative value indicates strong likelihood of poor server performance for a certain message sequence (e.g., *ADB*), the lower the negative value the worse. Sequences not observed in training are noted by a nil in the transition matrix. At the end of the training period, we choose the parameter with the least prediction error rate as our predictor (multiple parameters with relatively close error rates require human evaluation). To account for system initialization and warm up effects, we also construct a separate set of matrices for tracking the first N messages immediately following a system restart. At the end of the learning phase, we have a transition matrix that is fed to QMM. QMM evaluates the buffered messages before releasing a message to the head of the queue. The performance score is calculated by enumerating all possible orderings of the messages and for each ordering examine the message pairs and add the corresponding value from the transition matrix. A higher score indicates a sequence that is less likely to cause performance problems a low score indicates a sequence that is very likely to cause performance problems. The ordering with the highest score is chosen and the queue is ordered accordingly. QMM also performs this reordering after a server restart.

Table 1. A portion of the transition matrix from the resource leakage experiment

	A	B	C	D	E	F	...
AA	111	29	5	7	30	143	
AB	17	26	2	5	7	9	
AC	4	2	nil	2	2	12	
AD	2	-2	-1	nil	-1	5	
...							
BA	33	4	1	-1	3	36	
BB	11	3	2	-2	4	15	
BC	2	1	1	nil	1	-1	
...							

4 Overview of Sample Applications

The experimental evaluation of I-Queue uses data traces obtained from Worldspan, a leading GDS and the global leader in Web-based travel e-commerce. Each message is a request for pricing a travel itinerary. Through contractual agreements with its customers, Worldspan needs to generate a reply message within a predefined time limit. It has been observed in the new server that it will occasionally slow down and fail to meet its delivery deadlines. To emulate this behavior, we utilize the Worldspan traces and build simple models of applications servers. In this paper, we report results obtained from experimenting with two server models, both based on known memory leak behaviors as well as other resource leak problems.

4.1 Basic Server Model

The specific subsystem managed by I-Queue is Worldspan pricing query service. The service is handled by a farm of 1500 servers. Query messages from various clients are placed in one of two global queues. Each server in the farm acts independently of the others. As a server becomes available for processing, it pulls a message from the queue, processes the message, and generates a corresponding response message forwarded to other parts of the system for further processing. The request message contains a set of alternative itineraries for which the lowest available fare is to be found by the server. The response message contains a list of fares for each itinerary sorted by fare. All request messages are independent, and the server should maintain an average memory usage level when idle.

4.2 Experimental Models

The I-Queue implementation built for the pricing server monitors server behavior and correlates it with request sequences. To evaluate it, we construct two models in our labs and apply the traffic traces obtained from Worldspan to both of these models. The goal is to evaluate the I-Queue approach with simple failure models using realistic traces. Our future work will evaluate the approach with Worldspans actual server (see the Conclusion section for more detail).

A class of failures used to evaluate I-Queue assumes a server with a small memory leak that is directly proportional to the size of the input message. The larger the input message, the more itineraries to process and hence, more work by the server which can lead to a larger leak. Memory leaks cause gradual degradation in server performance due to memory swapping and can eventually result in a server crash. To detect problems like these, the I-Queue prototype implements an early detection module to detect performance degradation early. The module utilizes the Sequential Partial Probability Test (SPRT) statistical method for testing process mean and variance. The server model is reset when SPRT raises an alarm indicating performance degradation significant enough to be detected. Real-time SPRT was developed in the 1980s based on Wald's original process control work back in 1947 [19]. SPRT features user-specified type I and type II error rates, optima detection times, and applicability to processes with a wide range of noise distributions. SPRT has been applied in enterprise systems for hardware aging problems [20] and for other anomaly detection [21].

5 Experimental Results

Experiments were run in Georgia Tech's enterprise computing laboratory, the model server were built in Java and run on an x345 IBM server (hostname: dagobah), a dual 2.8GHz Xeon machine with 4GB memory and 1GB/s NIC, running RedHat Linux kernel version 2.4.20. Sensitivity analysis and queue management code were also implemented in Java.

Our first experiment concerns sensitivity analysis using Worldspan's traffic traces. We model a server with a minor bug that leaks memory in proportion to the input message size. Figure 3 shows the results of our detection algorithm. The x-axis shows the different parameters we analyzed, the corresponding error rate is plotted on the y-axis. The error rate is a measure of the quality of a specific parameter as a predictor with lower error rates indicating a better predictor. As seen in the graph, the parameter MSG-SIZE has error rate of 0% which means it accurately predicts the failure 100% of the time. In the second part of the experiment, we engage the queue management module to reorder the messages. Figure 3 shows the reduction in number of server restarts as a function of the buffer length in our managed queue. We observe here that we do not get a significant improvement with larger buffer sizes. Instead, a buffer size of 5 is sufficient for giving us adequate results. The training phase for our system involves running a batch of messages and observing the system behavior for them. A training set is composed of 460 messages and in the real server environment, a message typically takes 4-16 seconds to process. Thus, in the best case scenario, we need 30 minutes to train the system (not counting the time needed to re-start the server). Given the cost of the learning phase of our system, we next evaluate the effectiveness of our algorithms for different training set sizes. Figure 4 shows system improvement measured as average reduction in number of crashes on the y-axis versus number of training sets on the x-axis. The training sets are generated by random re-ordering of the original set. The reduction rate is measured by counting the number of server restarts for the original batch

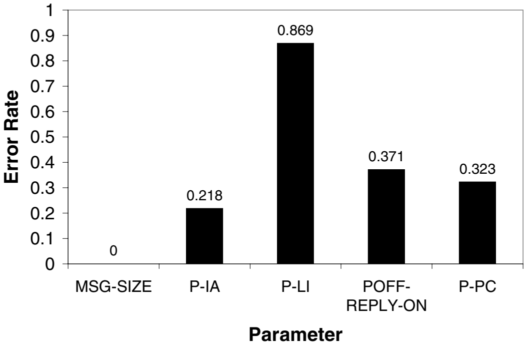

Fig. 3. Memory Leak Model: Sensitivity to various message parameters and message size

of messages with the managed vs. the unmanaged queue. It is a measure of the reduction in server faults we can achieve by using I-Queue, hence a higher reduction rate indicates more value in using I-Queue. The graph shows that we can get very good results with only a few training sets. This shows that I-Queue can be deployed with reasonable training time.

For brevity, we elide a second set of experimental results with a more complex failure model, termed the 'Connect Leak Model'. See [22] for more detail.

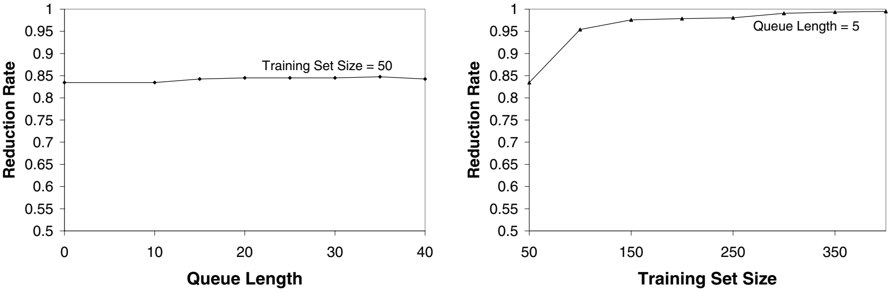

Fig. 4. Memory Leak Model: Error reduction measured for different queue length settings(left) and for different training set sizes (right)

6 Related Work

Our approach builds on established practice, in which machine learning techniques have been applied successfully to server and process monitoring. Application traces for detecting application faults are examined in [23,7,24,25]. These studies use application traces to detect a problem as it occurs and to recover the system by restarting the whole or parts of the system. Our approach differs in that we use application-defined methods to interpose and reschedule the message stream to minimize the number of system restarts and hence, increase system utility.

Our work aims to understand dynamic server behavior that results from client requests and then uses that knowledge to manage abnormal behaviors resulting from specific client request patterns. The approach can be embedded in Web service implementations, such as Apache Axis, to help build more reliable systems composed of web services. In that sense, we also complement other work in the area of dynamic web service composition [26,27].

The parallel computing domain has an extensive body of work on reliability using various monitoring, failure prediction [28], and checkpointing techniques [29]. Our work studies enterprise applications, specifically those in which system state is typically preserved in an external persistent storage (e.g., a relational database). In such systems, checkpointing the system state amounts to persisting the input event until it is reliably processed, and the cost of failures is dominated by process startup and initialization. In such environments, a reduction in the frequency of failures provides a tangible improvement to the system operators.

Additionally, the dynamic models we build (including the failure predictor) can prove valuable to system programmers as they try to troubleshoot the source of failure.

7 Conclusions and Future Work

This paper demonstrates a useful technique for automatically (1) detecting undesirable (i.e., poison) message sequences and then, (2) applying application-specific methods to achieve improved system performance and reliability. Future work includes conducting on-site experiments with the Worldspan search engine. We anticipate having more complex behaviors corresponding to multiple failure models interrelated in non-linear ways. We plan to approach this problem by using some of the well-studied clustering techniques (e.g., K-means analysis) to isolate the different behaviors and then apply our methods to each one separately.

Our longer term agenda is to use the monitoring and reliability techniques demonstrated in this paper in the context of service-oriented architectures. We are particularly interested in dynamically composed systems where users can create ad-hoc flows such as portal applications for high level decision support systems. In such systems, it is imperative to build middleware infrastructure to detect abnormal behaviors induced by certain component or service interactions and also, to impose 'firewalls' that can contain such behaviors and prevent them from further spreading into other parts of the system.

Acknowledgements. We gratefully acknowledge the help of James Miller in understanding the structure and parameters of the query messages. Many thanks also to Zhongtang Cai for directing us to the SPRT papers and algorithm.

References

1. Barham, P.T., Dragovic, B., Fraser, K., Hand, S., Harris, T.L., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003), Bolton Landing, NY (2003) 164–177
2. Agarwala, S., Poellabauer, C., Kong, J., Schwan, K., Wolf, M.: System-level resource monitoring in high-performance computing environments. Journal of Grid Computing **1** (2003) 273 – 289
3. IBM: Common base event. `http://www.ibm.com/developerworks/library/specification/ws-cbe/` (2003) [online; viewed:5/24/2006].
4. Swint, G.S., Jung, G., Pu, C., Sahai, A.: Automated staging for built-to-order application systems. In: Proceedings of the 2006 IFIP/IEEE Network Operations and Management Symposium (NOMS 2006), Vancouver, Canada (2006)
5. IBM: IBM Tivoli monitoring. (`http://www.ibm.com/software/tivoli/products/monitor/`) [online; viewed: 5/24/2006].

6. Bodic, P., Friedman, G., Biewald, L., Levine, H., Candea, G., Patel, K., Tolle, G., Hui, J., Fox, A., Jordan, M.I., Patterson, D.: Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In: ICAC '05: Proceedings of the Second International Conference on Automatic Computing, Washington, DC, USA, IEEE Computer Society (2005) 89–100
7. Roblee, C., Cybenko, G.: Implementing large-scale autonomic server monitoring using process query systems. In: ICAC '05: Proceedings of the Second International Conference on Automatic Computing, Washington, DC, USA, IEEE Computer Society (2005) 123–133
8. Mansour, M.S., Schwan, K.: I-RMI: Performance isolation in information flow applications. In Alonso, G., ed.: Proceedings ACM/IFIP/USENIX 6th International Middleware Conference (Middleware 2005). Volume 3790 of Lecture Notes in Computer Science., Grenoble, France, Springer (2005)
9. Keller, A., Ludwig, H.: The WSLA framework: Specifying and monitoring service level agreements for web services. J. Netw. Syst. Manage. **11** (2003) 57–81
10. Chen, M., Kiciman, E., Fratkin, E., Brewer, E., Fox, A.: Pinpoint: Problem determination in large, dynamic, internet services. In: Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track), Washington D.C. (2002)
11. Jin, W., Chase, J.S., Kaur, J.: Interposed proportional sharing for a storage service utility. In: Proceedings of the joint international conference on Measurement and modeling of computer systems, ACM Press (2004) 37–48
12. Candea, G., Cutler, J., Fox, A.: Improving availability with recursive microreboots: a soft-state system case study. Perform. Eval. **56** (2004) 213–248
13. Kumar, V., Cai, Z., Cooper, B.F., Eisenhauer, G., Schwan, K., Mansour, M.S., Seshasayee, B., Widener, P.: IFLOW: Resource-aware overlays for composing and managing distributed information flows. In: Proceedings of ACM SIGOPS EUROSYS'2006, Leuven, Belgium (2006)
14. Sun Microsystems: Java message service (JMS). (`http://java.sun.com/products/jms/`) [online; viewed: 5/24/2006].
15. Tibco: Tibco Rendezvous. (`http://www.tibco.com/software/messaging/rendezvous.jsp`) [online; viewed: 5/24/2006].
16. Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14** (1999) 54–62
17. Hanson, J.E., Whalley, I., Chess, D.M., Kephart, J.O.: An architectural approach to autonomic computing. In: Proceedings of the First International Conference on Autonomic Computing (ICAC'04), Washington, DC, USA, IEEE Computer Society (2004) 2–9
18. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. (1990) 267–296
19. Wald, A.: Sequential Analysis. John Wiley & Sons, NY (1947)
20. Cassidy, K.J., Gross, K.C., Malekpour, A.: Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In: DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, Washington, DC, USA, IEEE Computer Society (2002) 478–482
21. Gross, K.C., Lu, W., Huang, D.: Time-series investigation of anomalous CRC error patterns in fiber channel arbitrated loops. In Wani, M.A., Arabnia, H.R., Cios, K.J., Hafeez, K., Kendall, G., eds.: ICMLA, CSREA Press (2002) 211–215

22. Mansour, M.S., Scwhan, K., Abdelaziz, S.: I-Queue: Smart queues for service management. Technical Report GIT-CERCS-06-11, CERCS (2006)
23. Fox, A., Kiciman, E., Patterson, D.: Combining statistical monitoring and predictable recovery for self-management. In: WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, New York, NY, USA, ACM Press (2004) 49–53
24. Lohman, G., Champlin, J., Sohn, P.: Quickly finding known software problems via automated symptom matching. In: ICAC '05: Proceedings of the Second International Conference on Automatic Computing, Washington, DC, USA, IEEE Computer Society (2005) 101–110
25. Jiang, G., Chen, H., Ungureanu, C., Yoshihira, K.: Multi-resolution abnormal trace detection using varied-length n-grams and automata. In: ICAC '05: Proceedings of the Second International Conference on Automatic Computing, Washington, DC, USA, IEEE Computer Society (2005) 111–122
26. Wohlstadter, E., Tai, S., Mikalsen, T.A., Rouvellou, I., Devanbu, P.T.: GlueQoS: Middleware to sweeten quality-of-service policy interactions. In: ICSE, IEEE Computer Society (2004) 189–199
27. Tai, S., Khalaf, R., Mikalsen, T.A.: Composition of coordinated web services. In: Proceedings ACM/IFIP/USENIX International Middleware Conference (Middleware 2004). Volume 3231 of Lecture Notes in Computer Science., Toronto, Canada, Springer (2004) 294–310
28. Li, Y., Lan, Z.: Exploit failure prediction for adaptive fault-tolerance in cluster computing. In: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), Los Alamitos, CA, USA, IEEE Computer Society (2006) 531–538
29. Coffman, E., Gilbert, E.: Optimal strategies for scheduling checkpoints and preventative maintenance. IEEE Trans. Reliability **39** (1990) 9–18