Fault-Tolerant Middleware and the Magical 1%*

Tudor Dumitraş and Priya Narasimhan

Carnegie Mellon University, Pittsburgh PA 15213, USA
tdumitra@ece.cmu.edu, priya@cs.cmu.edu

Abstract. Through an extensive experimental analysis of over 900 possible configurations of a fault-tolerant middleware system, we present empirical evidence that the unpredictability inherent in such systems arises from merely 1% of the remote invocations. The occurrence of very high latencies cannot be regulated through parameters such as the number of clients, the replication style and degree or the request rates. However, by selectively filtering out a "magical 1%" of the raw observations of various metrics, we show that performance, in terms of measured end-to-end latency and throughput, can be bounded, easy to understand and control. This simple statistical technique enables us to guarantee, with some level of confidence, bounds for percentile-based quality of service (QoS) metrics, which dramatically increase our ability to tune and control a middleware system in a predictable manner.

1 Introduction

Modern computer systems are perhaps some of the most complex structures ever engineered. Together with their undisputed benefits for human society, their complexity has also introduced a few side-effects, most notably the inherent unpredictability of these systems and the increasing tuning and configuration burden that they impose on their users. The fact that large distributed systems, even under normal conditions, can exhibit unforeseen, complex behavior stresses that we are facing a veritable "vulnerability of complexity".

In this paper, we examine the unpredictability of fault-tolerant (FT) middleware. Typically used for the most critical enterprise and embedded systems to mask the effect of faults and to ensure correctness, FT middleware has higher predictability requirements than most other systems. We naturally expect that faults, which are inherently unpredictable, will have a disruptive effect on the performance of the system. In this paper, we show that *even in the fault-free case it is impossible to enforce any hard bounds on the end-to-end latency of an FT CORBA application.*

Conventional wisdom about commercial-off-the-shelf (COTS) software is that well designed and implemented ORBs behave in a sufficiently predictable manner when running on top of certain real-time operating systems. However, recent studies have independently reported that maximum end-to-end latencies of

* This work has been partially supported by the NSF CAREER grant CCR-0238381, the DARPA PCES contract F33615-03-C-4110, and also in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University.

CORBA and FT-CORBA middleware can be several orders of magnitude larger than the mean values and might not follow a visible trend [1,2,3]. At [3], Thaker lists many examples of systems that produce few outliers which are several orders of magnitude larger than the average: operating systems (Linux, Solaris, TimeSys Linux), transport protocols (UDP, TCP, SCTP), group communication systems (Spread), middleware and component frameworks (TAO, CIAO, JacORB, JDK ORB, OmniORB, ORBExpressRT, Orbix, JBoss EJB, Java RMI), including our own MEAD system. Our first goal in this paper is to evaluate how much predictability we can obtain by carefully choosing a good configuration of FT middleware components (operating system, ORB, group communication package and replication mechanism) and to analyze statistically the distribution of end-to-end latencies in the resulting system.

Additionally, fault-tolerant middleware poses fundamental trade-offs between dependability, performance and resource usage. Tuning the trade-offs is a delicate and non-trivial task because in most cases, this re-calibration requires detailed knowledge of the system's implementation. We have previously advocated an approach called *versatile dependability* [4], which consists in providing high-level "knobs" to control the external properties, such as latency or availability, that are relevant to the end-users of the system. However, the unpredictability observed in COTS middleware systems poses a challenge for tuning these high-level properties, unless we can find a simple and efficient method for making deterministic guarantees based on the configuration of the system. To reliably predict the average and maximum latencies based on specific configuration parameters (*e.g.*, number of clients, request rates and sizes, replication styles and degrees), we discard the highest 1% of the measured latencies. Then, we can establish bounds on the 99^{th} percentile of the metric monitored (*e.g.*, "the end-to-end response time will be less than 2s in 99% of the cases"). For a large class of applications, the value added by bounding the maximum latencies instead of the 99^{th} percentile does not justify the increased efforts that are required to achieve it [5]; in this paper, our second goal is to quantify, through an extensive empirical exploration, the confidence that we can place on this kind of guarantees.

In summary, this paper makes two concrete contributions:

- We provide substantial evidence that, despite choosing a good system configuration (operating system, ORB, group communication package and replication mechanism), the remote invocation latencies of a fault-tolerant CORBA infrastructure are difficult to bound and control, regardless whether or not faults occur (Section 2);
- We show that, by filtering out 1% of the largest recorded latencies, we obtain a set of measurements with predictable properties (Section 3); we call this the "*magical 1%" effect*. This effect dramatically increases our ability to tune the system in a predictable manner by specifying QoS guarantees based on the 99^{th} percentile of the targeted metric (*e.g.*, latency, throughput, reliability).

2 System Configuration for Achieving Predictability

In setting up the test bed for our empirical exploration, we aimed to select a configuration for our FT middleware with some of the best open source components available. We also provisioned the experimental environment to create the most favorable conditions for achieving predictable behavior.

To ensure the reproducibility of our results, we our experiments ran on the Emulab test bed [6]. We use the MEAD (Middleware for Embedded Adaptive Dependability) system [7], currently under development at Carnegie Mellon University, as our FT middleware. MEAD provides transparent, tunable fault-tolerance to distributed middleware applications. The system is based on library interposition for transparently intercepting and redirecting system calls, and features a novel architecture with components such as a tunable replicator, a decentralized resource monitor and a fault-tolerance advisor, whose task is to identify the most appropriate configurations (including the replication style and degree) for the current state of the system. MEAD implements active and passive replication based on the extended virtual synchrony model [4]. This model mandates that the same events (which can be application-generated as well as membership changes) are delivered in the same order at all the nodes of the distributed system, but without enforcing any timeliness guarantees. The extended virtual synchrony guarantees are enforced by the Spread (v. 3.17.3) group communication toolkit [8]. We have carefully tuned Spread's timeouts for our networking environment, to provide fast, reliable fault detection and steady performance. MEAD runs under the TAO real-time ORB (v. 1.4), which provide excellent average remote invocation latencies [9,3]. Our Emulab experiment uses 25 hosts connected with a 100 Mbps LAN. Each machine is a Pentium III running at 850 MHz (1697 bogomips), with 256 MB of RAM; the operating system is Linux RedHat 9.0 with the TimeSys 3.1 kernel.[1]

Since our goal is to evaluate the predictability of this system configuration (and not to assess or compare the overall performance), we believe that a micro-benchmark specifically targeting this goal is the most suited for our purpose. We have developed a micro-benchmark that measures the behavior of the system for different styles and degrees of server replication, with up to 22 connected clients. Each client sends a cycle of 10,000 requests, pausing a variable amount of time (from 0 to 32 ms) between requests. The replies from the server can also vary in size (from 16 bytes to 64 Kbytes). Each one of the physical nodes is dedicated to a single client or server. There are no additional workloads and no additional traffic imposed on the LAN to avoid interference with the experiments.

Admittedly, all these constraints imposed on the test bed make our setup somewhat unrealistic. However, our purpose was to provision the system in the best possible way for achieving predictable behavior. Since even this configuration turns out to be unpredictable (as we show in Section 2.2), it is very unlikely that a real-life industrial installation will prove to be more deterministic.

[1] This kernel is fully preemptible, with protection against priority inversion, $O(1)$ task scheduling complexity, and a fine timer granularity. This allows us to simulate request arrival rates with sub-millisecond precision.

Versatile Dependability. An important architectural feature of the MEAD system is the provision of tuning knobs. Adopting an approach called versatile dependability [4], MEAD defines a hierarchy of low-level and high-level knobs. Low-level knobs control the internal fault-tolerant mechanisms of the infrastructure and typically correspond to discrete (*e.g.*, the degree of replication) or even non-countable sets (*e.g.*, replication styles). In contrast, high-level knobs regulate externally-observable properties (*e.g.*, latency, availability) that are relevant to the system's users, and they should ideally have a linear transfer characteristic, with unsurprising effects. High-level knobs control the QoS goals of the system, and they are implemented based on low-level knobs; however, for defining trustworthy high-level knobs, the measured QoS values must be deterministically linked to the settings of the low-level parameters.

2.1 Experimental Methodology

In our experiments, we vary the following low-level parameters:

- *Replication style*: either active or warm passive replication;
- *Replication degree*: 1, 2 or 3 server replicas;
- *Number of clients*: 1, 4, 7, 10, 13, 16, 19 or 22 clients;
- *Request arrival rate*: the clients insert a pause of 0 ms, 0.5 ms, 2 ms, 8 ms or 32 ms. The lack of a pause (0 ms) represents bursty client activity;
- *Size of the reply messages*: 16 bytes, 256 bytes, 4 Kbytes or 64 Kbytes.

We have tested all 960 possible combinations of these parameters, collecting 9.1 Gbytes of data over a period of two weeks.[2] We statistically analyze this raw data to determine the probability distributions, the means, medians, standard deviations, the maximum and minimum values as well as the 99^{th} percentiles, the confidence intervals and the numbers and sizes of the outliers.

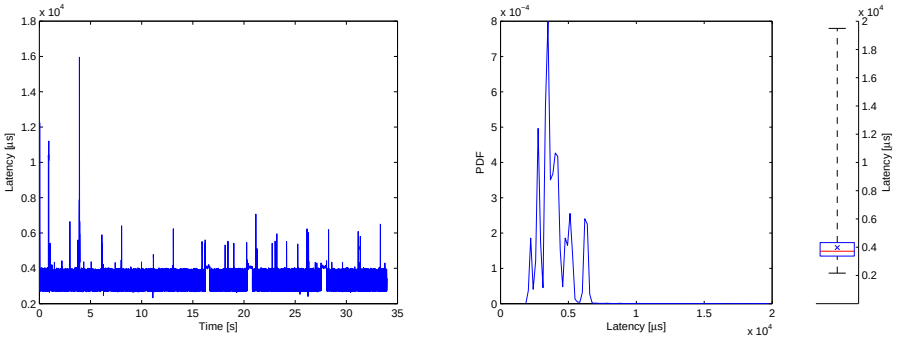
2.2 Evidence of Unpredictability

Figure 1(a) shows the raw end-to-end latencies (as measured on the client side) for a configuration with an unreplicated server and 4 clients, each sending 16 byte requests at the highest possible rate (no pause between requests). A few spikes that are much larger than the average are clearly noticeable in the figure. The effect of these spikes can be further analyzed by looking at Figure 1(b), which shows the probability density function for the end-to-end latencies. The distribution has a long tail, which indicates that there are a few samples, accounting for a small percentage of the measured values, that are much larger than the rest. The distribution is skewed only to the right because latency cannot take arbitrarily low values.[3] The same information can be represented in a more concise way by the "box and whisker" plot on the right side of the figure.[4] From the box plot, we can tell that the distribution is skewed to the right

[2] The full trace is available online at `www.ece.cmu.edu/~tdumitra/MEAD_trace`.

[3] MEAD's latency is bounded by the round-trip time of a regular TAO invocation.

[4] The box represents the size of the inter-quartile range (the difference between the 75^{th} and 25^{th} percentiles of the samples in the data set), while the whiskers indicate the maximum and minimum values. The mean and the median of the samples are represented by the line and the cross inside the box.



(a) End-to-end latencies (b) PDF of the measured latencies

Fig. 1. Unpredictable behavior of end-to-end latency.

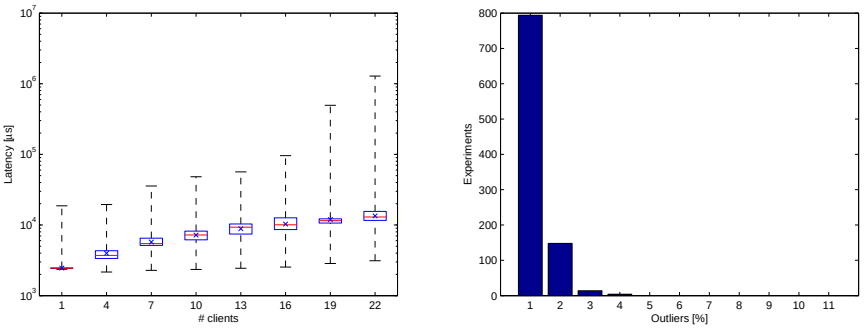
because the median is closer to the lower end of the box and the upper whisker is much longer that the lower one. But the most striking detail is the extent to which the maximum value exceeds most of the measured samples.

Such distributions are hard to characterize with a single metric. Mean values can describe well the average expected latency, but give no indication of the jitter. Maximum values are hard to use because they are largely unpredictable. Furthermore, the spikes seem to come in bursts, which breaks the defenses of many fault-tolerant systems which assume only single or double consecutive timing faults. This difference seems to be aggravated by an increasing number of clients, but without revealing a clear trend. In Figure 2(a), we can see that the maximum values are much higher that the means and that they increase and decrease (note the exponential scale) in an uncorrelated way with respect to the number of clients. The very large latencies are seen for only a few requests.

To determine which measurements are exceptional, we use the 3σ statistical test: any sample that deviates from the mean with more than 3σ is considered an outlier (σ is the non-biased standard deviation error). Figure 2(b) shows that, in most of our 960 experiments, the number of outliers was under 1%, with a few cases registering up to 2%, 3%, or 4%. Table 1 shows a breakdown of the numbers of outliers recorded for each value of the five parameters that we varied in our

Table 1. Impact of each parameter on the number of outliers.

Replication Style	**Replication Degree**	**# Clients**	**Request Rate**	**Request Size**
Active: 55.12%	1 replica: 28.51%	1: 9.26%	0 ms: 21.37%	16 b: 13.77%
Passive: 44.88%	2 replicas: 34.47%	4: 15.06%	0.5 ms: 20.69%	256 b: 12.10%
	3 replicas: 37.02%	7: 13.35%	2 ms: 20.50%	4 Kb: 21.14%
		10: 14.77%	8 ms: 20.16%	64 Kb: 52.99%
		13: 14.83%	32 ms: 17.28%	
		16: 12.29%		
		19: 9.24%		
		22: 9.21%		



(a) Increasing the number of clients. (b) The percentage of outliers.

Fig. 2. Empirical results.

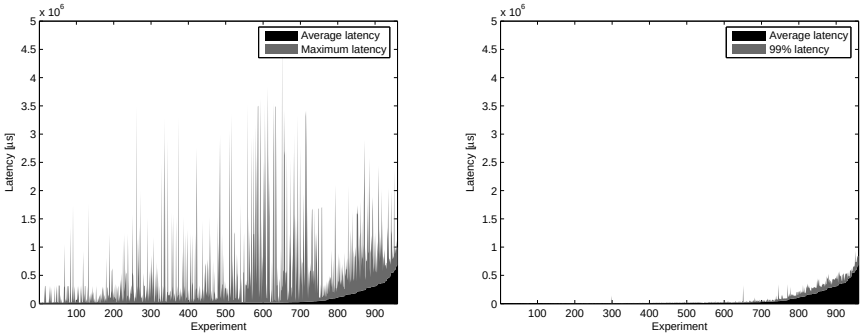
experiments. We notice that the outliers are almost uniformly distributed for all the values of these parameters (with the exception of the request size, where the 64 Kb messages produced more than half of the high latencies). This emphasizes that it is impossible to remove the outliers based on the system configuration and, knowing that outliers can be several orders of magnitude greater than the average (as shown above), we conclude that a sample of measured latencies that includes these outliers will have unpredictable maximum values.

3 The Magical 1%

Figure 2(b) shows that, in most cases, eliminating only 1% of the latencies would make all the remaining samples conform to the 3σ test. We now investigate what happens in our experiments if we remove 1% of the samples, hoping that we have isolated enough outliers to make the behavior of the system predictable. Figure 3(a) shows the high discrepancy between the means and the maximum values measured in all the experiments (in the figure, the experiments are sorted by the increasing average latencies). Note that it is hard to find a correlation between these two metrics, especially because the maximum values seem to be randomly high. If we remove the largest 1% from all these cases and we plot the data again, we get the "haircut" effect displayed in Figure 3(b): the randomness has disappeared and the 99^{th} percentiles do not deviate significantly from the average values. Only 1% of the measured end-to-end request latencies are responsible for the unpredictable behavior; discarding this 1% when making QoS guarantees results in a far more predictable system.

3.1 Expressing the Performance QoS Goals

The magical 1% is a simple, yet powerful approach for expressing QoS goals. While providing performance guarantees based on some sort of compliance with the 3σ test is quite difficult (because the test requires all the samples), using the magical 1% approach leads to specifications of QoS objectives that guarantee that 99% of the measured samples will fall within the stipulated bounds. Such percentile-based guarantees are: (i) easy to understand and relevant to the



(a) Before (b) After

Fig. 3. The "haircut" effect of removing 1% of the outliers.

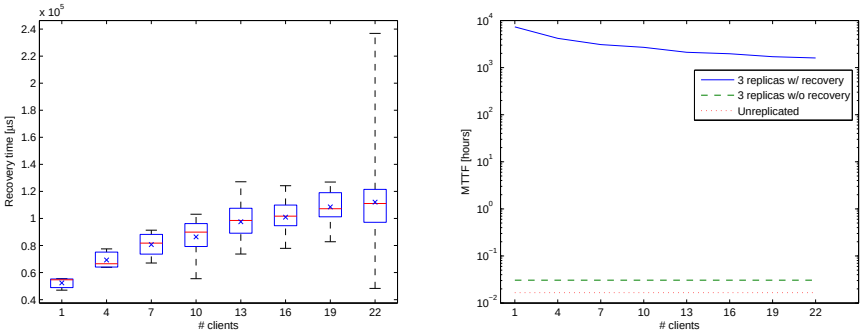
clients, since they reflect both the average and the sub-optimal behaviors, and (ii) easy to implement by service providers because the behavior in this case is predictable, as Figure 3 suggests. Other percentiles may be used (90% and 95% are quite common in the industry [5]) if more slack is desired, but for our fault-tolerant middleware system 99% appears to work effectively.

Outlier elimination should not be haphazardly performed: *when estimating the percentile-based bounds for a certain performance metric, we must take into consideration the semantics of that metric.* Most metrics will have outliers on only one side of the distribution (leading to a skewed distribution as shown in Figure 1(b)) because there are natural bounds which prevent them from extending indefinitely in the opposite direction; however, the side where the real outliers lie depends on the specific metric. For example, end-to-end latencies cannot be arbitrarily small because they are bounded by the underlaying network delays; therefore, outliers will be on the side of high end-to-end latencies. Conversely, for throughput the outliers will be on the lower side because throughput cannot become arbitrarily high. The magical 1% should only be trimmed on the side that can become unbounded for the corresponding metric.

From low-level to high-level knobs. To implement a latency-tuning knob, we must establish the connection between the low-level parameters and the 99^{th} percentile of the round-trip times. Our experiments reveal three trends: (i) latency increases linearly with the number of clients connected, as shown in Section 2.2; (ii) the latency decreases slightly with lower request rates; (iii) the latency increases faster than linearly with growing reply sizes. The first and second effects occur because less clients and reduced request rates alleviate the load of the server; the severe dependence of the latency on the request size is likely the result of additional work done by the operating system and by MEAD in the fragmentation and reassembly of large messages.

3.2 Expressing the Dependability QoS Goals

Dependable systems should mask as many faults as possible without resulting in failures. The reliability of the system is the probability of correct service for a given duration of time [10]. A common metric used for representing reliability is



(a) Recovery time. (b) $MTTF$ for 1 fault/minute.

Fig. 4. Using active replication with recovery to improve the system reliability.

the mean time to failure (MTTF). MEAD automatically restarts crashed replicas for improving the overall MTTF. The time between the restart and the instant when the new replica processes the first message is called the recovery time. Figure 4(a) shows the distributions of recovery times for a system with 3 replicas in the active replication style, with a request period of 2000 μs and a reply size of 4Kbytes. The recovery time depends on the number of clients connected, because the recovery algorithm must transfer the state and all the backlog of requests to the new replica. Figure 4(a) seems to indicate that recovery times are not as unpredictable as the response times, except for the case with 22 clients (compare with Figure 2(a)). This is a relatively surprising result, as the recovery process needs to perform several remote invocations to update the state of the new replica. Further investigation is needed to determine whether the "magical 1%" would be useful for making reliability guarantees.

We can convert the recovery times into mean times to failure using the formula [10]: $MTTF_{\text{3 replicas, restart}} = \frac{2\mu^2+7\mu\lambda+11\lambda^2}{6\lambda^3}$, where λ is the fault arrival rate and the recovery rate μ is the inverse of the mean recovery time. For example, at a constant rate of 1 fault per minute (which is unrealistically high), using active replication style with 3 replicas will add only 50 seconds to the $MTTF$; using the recovery strategy will raise the $MTTF$ up to 7313 hours (about 305 days). This is shown in Figure 4(b).

4 Discussion

The source of the apparently random behavior reported here cannot be easily isolated to the operating system, the network, the fault-tolerant infrastructure, the ORB, or the application. In fact, most operating systems exhibit seemingly unbounded latencies – at least for network operations, but in some cases for IPC and scheduling as well [3]. Furthermore, even systems built on top of the best designed, hard real-time operating operating systems manifest the same symptoms when combined with off-the-shelf ORBs and group communication packages. This behavior is likely the result of combining COTS components that: (i) were not built and tested jointly, and (ii) were designed to optimize for the common case among a wide variety of workloads, rather than to enforce tight bounds for the worst-case behavior.

Breaking the Magical 1%. Our empirical observations are clearly specific to the configuration under test. While the unpredictability of maximum end-to-end latencies has been reported for many different systems [3], the effectiveness of our "magical 1%" approach remains to be verified for other settings. So far, we have not tried running the application on a wide area network, using a different OS, working in a an environment with intermittent network connectivity (*e.g.*, a wireless network), simulating flash crowds, having other workloads that compete for CPU time and produce interfering traffic, or using virtual, rather than physical computing resources. Finally, certain applications (*e.g.*, embedded systems) will not be able to use percentiles; in such cases, nothing short of predictable worst-case behavior will be sufficient to ensure safety.

5 Related Work

The Fault Tolerant CORBA standard [11] specifies ten parameters that can be adjusted for achieving the required levels of performance and fault-tolerance for every application object, but it does not provide any insight on how these parameters should be set and re-tuned over the application's lifetime [12]. Even for fixed values of these parameters, the end-to-end latencies are hard to bound because they have skewed and sometimes bimodal distributions [2] (a phenomenon we have also observed). For the CORBA Component Model, it has been noted that a small number of outliers (typically less than 1%) causes maximum latencies to be much larger than the averages [1]. Thaker [3] reports that many systems produce a few numbers of outliers several orders of magnitude larger than the average: operating systems (Linux, Solaris, TimeSys Linux), transport protocols (UDP, TCP, SCTP), group communication systems (Spread), middleware and component frameworks (TAO, CIAO, JacORB, JDK ORB, OmniORB, ORBExpressRT, Orbix, JBoss EJB, Java RMI), including our own MEAD system. A percentile-based approach for specifying QoS guarantees is a common practice in the IT industry for most systems outside the real-time domain [5]. In this paper, we evaluate the virtues of such a percentile-based approach, with an emphasis on extracting tunability out of complex systems, rather than simply a risk-mitigation approach for service providers.

6 Conclusions

In this paper, we examine the predictability of a fault-tolerant, CORBA-compliant system. We try to achieve predictable behavior by selecting a good system configuration, but we show that, for almost all 960 parameter combinations tested, the measured end-to-end latencies have skewed distributions, with maximum values several orders of magnitude larger than the averages. These high latencies are due to a few (usually less than 1%) outliers which tend to come in bursts. The number of clients, the replication style and degree or the request rates neither inhibit nor augment the number of outliers. While the exact causes for this unpredictability are hard to pinpoint in every case, this seems to be a generic side-effect of complexity and of system design goals that focus on optimizing the average behavior (rather than bounding the worst case).

We also present strong empirical evidence of a "magical 1%" effect: by removing 1% of the highest measured latencies for each configuration, the remaining samples have more deterministic properties. We show that the 99^{th} percentile follows the trend of the mean and that it can be used for making latency guarantees. The significance of this result is that it allows us to extract tunable, predictable behavior (with respect to performance and dependability) out of fairly complex, unpredictable systems. While this percentile-based guarantees are clearly inappropriate for hard real-time systems, they can be of immense benefit to enterprise service providers and customers, who want reasonable, quantifiable and monitorable assurances. Since similar behavior has been reported for many other systems, we believe that our "magical 1%" opens an interesting avenue for further research in statistical approaches for handling unpredictability.

Acknowledgments. The authors would like to thank David O'Hallaron, Asit Dan, Daniela Roşu, Jay Wylie and the anonymous reviewers for their invaluable suggestions and ideas related to this topic.

References

1. Krishna, A.S., Wang, N., Natarajan, B., Gokhale, A., Schmidt, D.C., Thaker, G.: CCMPerf: A benchmarking tool for CORBA Component Model implementations. The International Journal of Time-Critical Computing Systems **29** (2005)
2. Zhao, W., Moser, L., Melliar-Smith, P.: End-to-end latency of a fault-tolerant CORBA infrastructure. In: Object-Oriented Real-Time Distributed Computing, Washington, DC (2002) 189–198
3. http://www.atl.external.lmco.com/projects/QoS/.
4. Dumitraş, T., Srivastava, D., Narasimhan, P.: Architecting and implementing versatile dependability. In de Lemos, R. et al., ed.: Architecting Dependable Systems III. Lecture Notes in Computer Science. Springer-Verlag (2005)
5. Alistair Croll: Meaningful Service Level Agreements for Web transaction systems. LOOP: The Online Voice of the IT Community (2005)
6. White, B. et al.: An integrated experimental environment for distributed systems and networks. In: Symposium on Operating Systems Design and Implementation, Boston, MA (2002) 255–270
7. Narasimhan, P., Dumitraş, T., Paulos, A., Pertet, S., Reverte, C., Slember, J., Srivastava, D.: MEAD: Support for real-time, fault-tolerant CORBA. Concurrency and Computation: Practice and Experience **17** (2005) 1527–1545
8. Amir, Y., Danilov, C., Stanton, J.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: International Conference on Dependable Systems and Networks, New York, NY (2000) 327–336
9. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time Object Request Broker. Computer Communications **21** (1998) 294–324
10. Siewiorek, D., Swarz, R.: Reliable Computer Systems. 2 edn. Digital Press (1992)
11. Object Management Group: Fault Tolerant CORBA. OMG Technical Committee Document formal/2001-09-29 (2001)
12. Felber, P., Narasimhan, P.: Experiences, approaches and challenges in building fault-tolerant CORBA systems. IEEE Transactions on Computers **54** (2004) 497–511